Entwurfskonzept

Update zum Release v0.5 (Fertiges Release v1.0)

Projektgruppe swp10-08:

Victor Christen
Anastasiya Chyhir
Moritz Kähler
Klaus Lyko
Marcello Scrobanita
Martin Türpe

Inhaltsverzeichnis

1 Allgemeines

Der Comorbidity Viewer ist eine auf Java basierende Applikation zur visuellen Modellierung von Human-Disease-Networks.

2 Produktübersicht

Krankheiten und deren Beziehungen, d.h. hier vor allem komorbides Auftreten, werden in Form eines Graphen visuell zugänglich gemacht. Die Benutzeroberfläche besteht aus drei Hauptkomponenten: Einem Panel zur Steuerung der Anwendung, dem Panel auf dem der Graph gezeichnet wird und weiterhin einer Liste der gerade angezeigten Krankheiten. Ferner kann auf Wunsch eine Tabelle eingeblendet werden, die den grafisch dargestellten Graphen textuell zur genaueren Analyse der Krankheiten und Zusammenhänge zugänglich macht.

3 Grundsätzliche Struktur- und Entwurfsprinzipien des Systems

Die Hauptbestandteile der Anwendung werden mit den Möglichkeiten die das prefuse-Toolkit bietet verwirklicht. Dem folgend beruht die Hauptstruktur der Anwendung auf dem Designpattern dem die prefuse-Bibliothek unterliegt: das sog. *Information Visualization Reference Model (IVRM)*.

Für die Interaktion des Benutzters mit der Software über eine graphische Oberfläche, das *view*, wird das *Model-View-Controller* – Designpattern verwendet. Je nach Kommunikationsbedarf und Funktionen der einzelnen Komponenten, wird das Pattern in unterschiedlichen Varianten verwendet.

3.1 Aufbau des Systems

Demnach gliedert sich das Produkt in die 4 Hauptpakete: **data**, **visual_abstraction**, **view** und **controls**.

Weiterhin gibt es das Hauptpaket **db_connector**, der die Schnittstelle zwischen Quelldaten, Datenbank und der prefuse-internen Daten-Repräsentation darstellt.

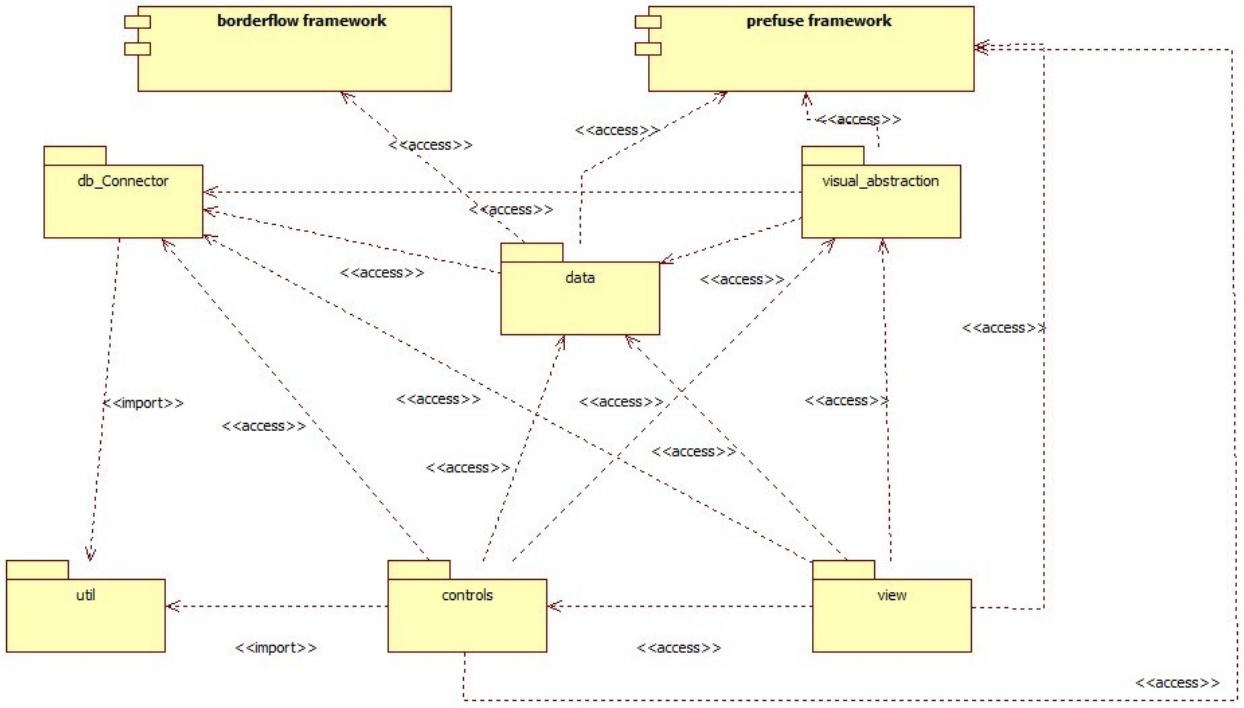
Dadurch wird eine Kapselung der verschiedenen Datenschichten und den betreffenden Klassen und Methoden, die auf diesen operieren, erreicht.

3.2 Generelle Entwurfsprinzipien

Um die Erweiterbarkeit des Systems zu erleichtern wurden an den entscheidenen Stellen Interfaces definiert, damit ist eine leichte Änderung und Anpassung des System an andere Anforderungen gewährleistet.

So wurde das Interface *DataReader* als Schnittstelle zwischen dem DBConncetor und dem Data-Paket definiert. So ist es theoretisch möglich das Produkt später auf eine Arbeit mit Dateien umzustellen.

3.2.1 Paketdiagramm



4 Grundsätzliche Struktur und Entwurfsprinzipien der einzelnen Pakete

4.1 Paket: db_connector

Dieses Paket dient als Schnittstelle zwischen den Quelldaten und den intern verwendeten Prefuse.Tables bzw. Graph. Da die Anwendung mit zum Teil umfangreichen Datenbeständen umgehen muss, wurde die Verwendung einer Datenbank als effiziente Lösung gewählt.

4.2 Aufgaben des Pakets

Der Comorbidity-Viewer setzt zur Verwendung eine vorhandene MySQL-Datenbank voraus. Die Verbindung zu dieser wird hier implementiert. Es wird dabei JDBC verwendet, was die Verwendung eines anderen Datenbanksystems erlaubt.
Es gibt zwei Hauptfunktionen des Systems, die hier bereitgestellt werden: Erstens gibt es eine Klasse, welche eine Erstellung eines entsprechenden Datenbankschemas (siehe unten) übernimmt. Außerdem wird hier auch das Einlesen der Rohdaten in entsprechende Tabellen der Datenbank übernommen. Diese liegen in Tab-separierten Textdateien, welche dem HuDiNe-Format entsprechen, vor (DataBaseConnector). Ferner wird standardmäßig eine Tabelle (icdcodes) erstellt, die eine Zuordnung der ICD-9-CM Codes zu den entsprechenden Namen bzw. Abkürzungen erlaubt (ICDFileReader).

Die zweite Hauptaufgabe besteht darin die Suche nach Kanten in den Quelldaten und deren Bereitstellung als Prefuse.Data.Table zu implementieren. Also konkret entsprechende Suchanfragen in SQL-Statements umzuwandeln und entsprechende Queries in der Datenbank auszuführen. Die verwendeten Queries, sowohl für diese als auch andere Anfragen stellt die Klasse ConstantQueries bereit. Hierbei werden die bereits in Prefuse vorliegenden Schnittstellen (im Paket data.io.sql) verwendet, die es ermöglicht ein ResultSet einer DB-Anfrage in eine Prefuse-Table zu speichern. Dies wird in der DataBaseReader Klasse implementiert.

4.3 Datenbankschema

Die Datenbank operiert im wesentlichen auf Kantentabellen. In diese werden aus der betreffenden Datei die zehn Spalten eingelesen. Da verschiedene Quelldateien installiert werden können, kann jeder Edge-Tabelle ein, den betreffenden Datensatz identifizierenden, Name gegeben werden.
Die bereits erwähnte Node – Tabelle (icdcodes) wird beim ersten Start des Programms in die Datenbank geladen. Das ermöglicht jeder Kante, der verschiedenen Edge-Tabellen die betreffenden Krankheitsnamen zuzuordnen.

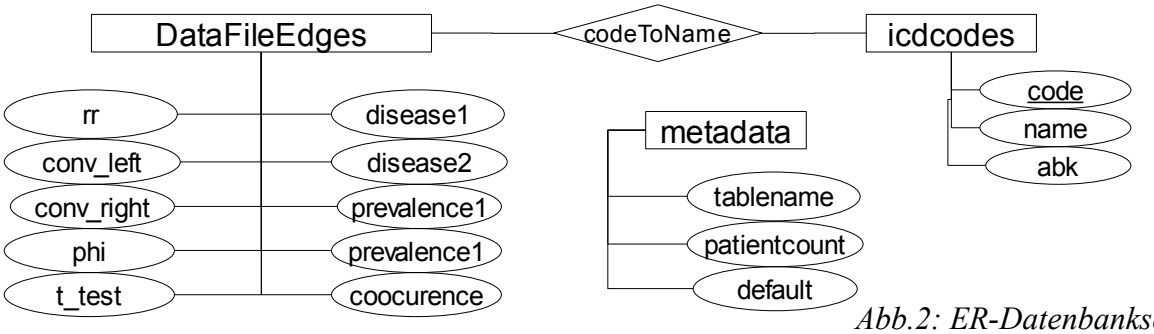


Abb.2: ER-Datenbankschema

Eine genauere Übersicht über die verwendeten Datenbank-Datentypen in folgendem Relationenmodell:

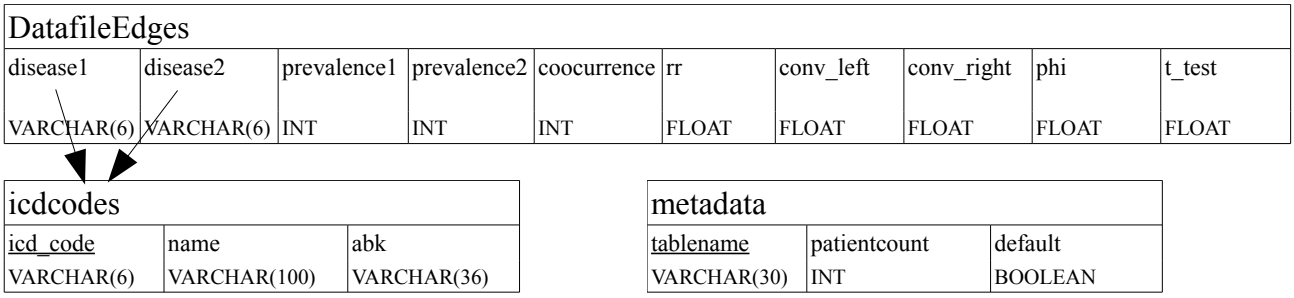


Abb. 3: Relationales Schema

Es wird eine Node-Tabelle erstellt und bei jeder Suchanfrage überprüft, ob die gesuchte Krankheit im ausgewählten Datansatz enthalten ist. Somit gibt es eine Node-Tabelle für alle Krankheiten.

4.4 Verantwortlichkeiten innerhalb des Pakets

Die Klasse *ICDFileReader* läd die Daten für die Node-Relation aus entsprechender (mit ausgelieferter allcodes.csv) Datei.

Die Klasse *DataBaseConnector* stellt eine Verbindung zur Datenbank her, generiert entsprechende SQL-Statements und stellt ggf. die ResultSets zur Verfügung. Genauer gesagt, werden die SQLQueries in der Klasse ConstantQueries generiert und als String dem DBConnector zur Verfügung gestellt. Das Paket beinhaltet auch eine Klasse *DBConstants* die zur Kapselung aller verwendeten Konstanten (wie zum Beispiel Spaltennamen der Tabellen) dient. Im DataBaseConnector wird auch die Methode *createEdgeTable(tableName, patientCount)* bereit gestellt. Somit können Tab-separierte Textdateien (im HuDiNe Format) in die Datenbank geladen werden. Der *DataBaseConnector* erbt von *Observable* , um die Synchronisation mit aufrufenden Komponenten zu erleichtern.

DBMetadata stellt Methoden bereit um mit verschiedenen Datansätzen in der Datenbank arbeiten zu können. Um dies zu ermöglichen wird beim Erstellen einer Datenbank automatisch eine *MetaData* Tabelle erstellt. In ihr werden alle vorhandene Datensätzen (Kantenlisten) mit Namen und Anzahl der untersuchten Patienten gespeichert. Die Methode *getAllTables* gibt alle vorhandenen Datensätze, mit Ausnahme der Tabellen *metadata* und *icdcodes*, zurück. Die zuletzt verwendete Tabelle lässt sich mit *getDefaultRelation* ermitteln. Entsprechende Setter sind vorhanden.

Die Klasse *DataBaseReader* implementiert das Inteface *DataReader*. Dies gewährleistet, dass Prefuse-Tabellen durch die get-Methoden übergeben werden. Diese Bereitstellug der Prefuse-Tables ist aufgrund der Transformierung eines ResultSets einer SqlQuery in eine Prefuse-Table realisierbar.

Details siehe *ComorbidityViewer-Design_dbConnector-detailiert.jpg*

4.5 Paket: data

GraphManager- Klasse:

Die zentrale Klasse dieses Pakets ist der GraphManager. Diese Klasse hält die Innstanz des Graphen, der zur kompletten Laufzeit des Programms existiert. Des Weiteren werden die Kantentabelle und die Knotentabelle gespeichert. Auf diesen Datenstrukturen werden alle Operationen wie das Löschen von Kanten und Knoten bzw. das Hinzufügen von diesen ausgeführt. Durch den von prefuse intern implementierten GraphListener wird der Graph bei jeder Veränderung seiner Kantentabelle und Knotentabelle aktualisiert. Die Tabellen halten neben dem Aufbau des Graphen weitere Informationen, die später mit angezeigt werden oder deren Bedeutung speziell als Größe des Knotens bzw. als Größe der Kante visualisiert wird. Durch die Methode `explore` kann der Graph vom angegebenen Knoten erweitert werden, dass heißt dem Graphen werden alle Tupel aus dem gewählten Datensatz hinzugefügt, die dem Mindestkantengewicht entsprechen. Die Tupel der

Knotentabelle werden aus der bestehenden Kantentabelle mit den spezifischen Knotenattributen z.B Prävalenz, Name, etc erstellt. Um die Kategorien zu ermitteln dient die Klasse *DiseaseCategories*. Bei Übergabe eines ICDCodes, wird ein Integer zurück geliefert, dessen Wert entscheidend für die richtige Zuordnung der Knotenfarbe zu den Kategorien ist, die aus einem Integer Array ermittelt werden kann.

Um die Daten zu akquirieren wird die *DataBaseReader*- Klasse verwendet. Mithilfe des Interfaces *GraphExplorer* wird sichergestellt das eine Preduse-Table vom *DataBaseReader* zurückgeliefert wird . Dadurch wird das Einfügen neuer Tupel in die bestehenden Tabellen aufgrund der gleichen Datenstruktur erheblich vereinfacht.Des Weiteren wird ein Austausch der Komponente, die für die Datengenerierung verantwortlich ist , gewährleistet. So ist eine Datenbank für die Funktionalität des Programmes nicht gefordert, wenn man einen adäquaten Ersatz fur die Datengenerierung implementiert, der ebendalls das Interface implementiert.

Wenn eine neue Krankheit als Graph visualisiert wird, muss der bestehende Graph geleert werden , damit kein inkonsistenter Graph entsteht.

Des Weiteren besitzt die GraphManager- Klasse Methoden,`setMinimumWeight(float weight)` und `setToPhiMode(boolean phiMode)`, um das aktuelle minimale Kantengewicht oder den Kantengewichtstyp zu ändern. Dabei ist darauf zu achten das der Kantengewichtstyp nur bei leeren Graphen neu gesetzt werden kann.

Für die GraphManager- Klasse wird ein Singleton-Pattern eingesetzt. Somit wird gewährleistet, dass alle Klassen, die mit dem Graphen arbeiten, auch mit dem selben Graphobjekt arbeiten und Methoden aufrufen können die auf den Tabellen des Graphen operieren. Da der GraphManager den Graph enthält, stellt diese Klasse den Kern der Anwendung dar. Damit sich alle Komponenten, die auf den Graphen angewiesen, auf die Änderungen des Graphen reagieren können, erweitert diese Klasse die Observable Klasse. Bei jeder Änderung bzw. vor jeder Änderung wird ein setChange() Flag gesetzt. Somit ist gewährleistet das alle darauf aufbauenden Klassen dementsprechend reagieren. Des Weiteren liegt hier auch ein Subpaket für das Clustern des Graphen. Dort werden die Knoten den einzelnen Clustern zugeordnet, sodass jeder Knoten des Teilgraphen seinem Cluster zu gewiesen wird. Verwendung findet dabei die Borderflow-Bibliothek.

Details siehe *ComorbidityViewer data.jpg*

4.6 Paket: visual_abstraction

Dieses Paket ist verantwortlich für das Mapping des Graphen, aus dem Data-Paket auf eine Visualisierungsinstanz, so dass der Graph später visualisiert werden kann. Des Weiteren wird die Darstellung des Graphen spezifiziert, dass heißt es werden Knotenlabels, Kantenstärke, Knotengröße, etc. festgelegt. Um die Anordnung der Knoten auf dem Display zu realisieren werden der Visualisierung verschiedene Layouts hinzugefügt. Die Darstellung des Cluster wird in diesem Paket implementiert.

4.7 Aufgaben des Paketes

Dieses Paktet enthält eine Klasse *CVVisualization*, die die Prefuse-Visualization erweitert.

Durch Hinzufügen des Graphen werden die graphspezifischen Daten mit der Visualisierung gemappt. Des Weiteren werden in dieser Klasse die Grundeinstellungen des Graphen definiert, das heißt Knotenbeschriftung, Knotengröße, Kantenstärke, Kantenbeschriftung etc.. Um zwischen den Beschriftungen der Knoten zu wählen, werden Methoden bereit gestellt, die es ermöglichen zwischen ICD-Code bzw. Krankheitsname zu wechseln, sowie das Ausschalten jeglicher Knotenbeschriftung. Die Beschriftung wird durch das Decorator- Inteface realisiert. Zum Umschalten der einzelnen Modi, werden lediglich die boolean Variablen, die verantwortlich sind für die Anzeige des ICDCodes oder des Namens bzw. das komplette Ausschalten der Sichtbarkeit der Decorators, neu gesetzt durch die setter-Methoden der *CVVisualization*-Klasse. Für die Beschriftung der Decorator sind die *DecoratorLabelLayout* – und die *DecoratorHoverLayout*- Klassen verantwortlich. Abhänging von dem Modi wird die Sichtbarkeit jedes *DecoratorItems* ausgestellt bzw. eingestellt oder mit einem ICDCode bzw. mit dem Namen beschrieben. Um die Knoten farblich nach Kategorien zu gestalten, erweitert die Klasse *NodeColorAction* die prefuse.Action. Diese Action färbt jeden Knoten, abhängig von seiner Kategorienummer, die im *GraphManager* gesetzt wurde.

Die Kanten werden je nach Kantengewicht unterschiedlich stark gezeichnet. Die Klasse *EdgeSizeAction* erweitert die ActionSwitch Klasse, um je nach Kantengewichtstyp die Kantenstärke auf Basis der Kantengewichte zu definieren. Da die *CVVisualization* die Basisdaten für die Visualisierung enthält, besitzt sie auch die verschiedenen Layouts. Da zu einem Zeitpunkt nur ein bestimmtes Layout angezeigt werden kann, gibt es eine Methode, die den Graphen dem ausgewählten Layout anpasst.

Jedes Layout wird in der dafür vorgesehenen Klasse implementiert. Da Prefuse die meisten Layouts enthält, werden diese von unseren Layout-Klassen als Instanz in einer ActionList gespeichert. Eine ActionList ist notwendig um jedem Layout eine RepaintAction() hinzuzufügen. Desweiteren werden dem *Hierarchical Layout* und dem *RadialHierarchicalLayout* eine *OrientationAction* hinzugefügt. Sie ist dafür verantwortlich, dass der Graph beim Verschieben und Explorieren (z.B beim *RadialHierarchicalLayout*) sich nicht an seinem ursprünglichen Ort zurück bewegt. Mögliche Layouts sind: *SpringLayout* (ForceDirectedLayout), *RadialHierarchialLayout* (RadialTreeLayout), *CVCircleLayout* (CircleLayout) und *HierarchialLayout* (NodeLinkTreeLayout).

Um später explorieren zu können müssen Methoden vorhanden sein, die die Actions, die für das Layout oder die Darstellung von Kanten und Knoten verantwortlich sind, ausschalten. Wenn die neuen Daten in die Kanten- bzw. Knotentabellen geladen wurden, werden die Actions durch entsprechende Methode wieder eingeschaltet. Dadurch ist eine Aktualisierung des visuellen Graphen auf die Daten der Kanten- und Knotentabelle der Graphinstanz aus dem Data-Paket gewährleistet.

Details siehe *ComorbidityViewer visual_abstraction.jpg*

4.8 Paket: view

Das view-Paket enthält alle Komponenten und Dialoge für die graphische Oberfläche des Comorbidity Viewers. Über diese Komponenten wird der Graph zusammen mit den dazugehörigen Informationen angezeigt. Das View ist mit einer Vielzahl von Interaktionselementen ausgestattet, über die sich der Nutzer weitere Informationen zum Netzwerk einholen und es weiter explorieren kann.

Für den Verbindungsaufbau und Erstellung einer Datenbank, sowie den Daten-Import geeigneter Dateien in die Datenbank, liefert die view den *DBDialog* und den *DataImportDialog*.

Zur Anzeige des Graphen, seiner Knoten- und Kanten-daten und zur interaktiven Exploration gibt es jeweils einzelne *Panels*, welche die einzelnen Aufgaben übernehmen. Diese Panels (*OptionPanel, ICDSearchPanel, NodePanel, EdgePanel, ClusterPanel, ICDCategoryPanel* und *InfoBar*) sind alle in das Haupt-Panel (*CVMainPanel*) eingebunden und bilden damit die Subkomponenten der Benutzeroberfläche.

Das Eventhandling wird einerseits von den Controllern aus dem Paket controls, andererseits aber auch intern, in sogenannten annonymen Klassen übernommen. Ersteres enspricht mehr dem MVC-Pattern und letzteres eher dem Swing- und prefuse-typischen MVP (Model-View-Presenter) – Design.

Aufgrund der Zentralisierung der einzelnen Interaktions-Komponenten auf dem MainPanel, soll die Software sowohl als stand-alone-Anwendung auf einem JFrame (dieser wird von *CVFrame* erweitert) als auch auf einem JApplet ausgeführt werden können. Der wesentliche Unterschied ist, dass beim Applet (*CVApplet*) nicht alle Funktionen zur Verfügung gestellt werden, denn dieses soll dann unter anderem auf einem zentralen Server laufen, auf die viele Benutzer zugreifen können. Das heißt hierfür wird z.B. die Daten-Import-Funktion nicht bereitgestellt.

4.8.1 Komponenten und Funktionen des Pakets

Das „View“ bildet mit seinen Komponenten die Schnittstelle zwischen Benutzer und Anwendung. Um die Software sowohl als Desktop-Applikation, wie auch als Applet benutzen zu können und die Funktionen sauber zu kapseln, wurde im Entwurf ein hierarchischer Aufbau der Komponenten gewählt:

Die „Trägerkomponente“ für die Stand-Alone-Anwendung ist die Klasse *CVFrame*, welche einen *JFrame* erweitert und für das Applet die Klasse *CVApplet*, die von *JPrefuseApplet* erbt. Auf diesen Trägern wird dann das *MainPanel* aufgebracht, welches die nach Funktion kategorisierten Subkomponenten enthält. Die Struktur und Funktionalität ist wie folgt:

- *MainPanel*
  - *CVDisplay* (erweitert *prefuse.Display*)
    - Anzeigekomponente für den Graphen / das Teilnetz
    - Funktionen:
      - Knotenauswahl, Knoten-Expansion (= GraphExploration)
      - Zoom, Verschieben / Scrollen
      - weitere Funktionen über die Toolbar auf dem *OptionPanel*
    - Position: oben im östlichen SplitPane
  - *NodePanel* (erweitert javax.swing.*JPanel*)
    - Beinhaltet die Knotentabelle des angezeigten Graphen
    - Funktionen:
      - Übersichtliche Auflistung der momentan angezeigten Krankheiten mit ICD-Code und Namen
      - Zugang zu weiterführenden Informationen über externe Links (3 Buttons für Links zu Wikipedia, DBpedia und der amerikanischen ICD9Data-Seite)
        - die URLs für diese Links werden durch die Klasse *LinkIntegration* im Paket **util** generiert und aufgerufen
    - Position: auf dem *NodePane* (westlich) in einem JTabbedPane zusammen mit dem *ClusterPanel*
  - *EdgePanel* (erweitert *javax.swing.JPanel*)
    - Beinhaltet die Kantentabelle des angezeigten Graphen
    - Funktionen:
      - Anzeige der Kantenliste mit allen zugehörigen Informationen aus dem Datensatz
      - Tabelle kann nach einer beliebigen Spalte sortiert werden
    - Position: auf dem östlichen SplitPane unten
  - *ClusterPanel* (erweitert *javax.swing.JPanel*)
    - Beinhaltet Liste (genaugenommen einen *JTree*) von aktuellen Clustern sowie Clustering-Optionen
    - Funktionen:
      - Übersichtliche Auflistung von Clustern in einem JTree
      - Clustering-Optionen (Clustering an / aus)
    - Position: in einem Tab auf dem *NodePane* unterhalb des *NodePanel*s
  - *OptionPanel* (erweitert *javax.swing.JPanel*)
    - Ist das Haupt-Interaktions-Element für die view. Es beinhaltet mehrere Toolbars, welche wiederum die Funktionen für das Zoomen, Layouten und Anzeigen von den DecoratorItems auf dem *CVDisplay* bereitstellen.
    - Funktionen:
      - Zoom, Layout und erweiterte Einstellungen für den Graphen (Anzeige von DecoratorItems)
      - Anzeige und Auswahl von den Datensätzen der gewählten Datenbank aus einer ComboBox
      - Ein- und Ausblenden des *Node*- und *Edge-Panes* (das *NodePane* ist der komplette westliche SplitPane-Teil, welcher die TabbedPanes mit weiteren Subkomponenten enthält; das EdgePane liegt südlich auf dem Haupt-SplitPane, welches die view vertikal trennt, also in eine linke und eine rechte Seite)
      - Aktivierung des *DB*- und des *DataImportDialog*es
    - Position: nördlich direkt unter der Menubar
  - *ICDSearchPanel* (erweitert *javax.swing.JPanel*)
    - Panel mit Suchfunktion für Krankheiten (zur Auswahl einer Krankheit, um ein *neues* Teilnetz über dem gewählten Datensatz zu explorieren)
    - Funktionen:
      - Suche nach Krankheiten per ICD-Code oder Schlagwörtern im geladenen ICD-Datensatz (ICD-9-CM ist standardmäßig installiert)

        - Auflistung der Suchergebnisse und bei Auswahl Exploration eines neuen Graphen in einer Tabelle
        - Es kann ein Suchergebnis aus der Tabelle ausgewählt werden, und nach "phi" oder "rr" mit einem beliebigen Wert größer 0 exploriert werden
    - Position: in einem Tab auf dem *NodePane*
- *ICDCategoryPanel* (erweitert *javax.swing.JPanel*)
    - Legende für die Zuordnung von Knotenfarben zu Krankheitskategorien
    - Position: in einem Tab auf dem *NodePane*
- *InfoBar* (erweitert *javax.swing.JPanel*)
    - Anzeige diverser Status-Informationen: Datenbank-System-URL (hostname), gewählte Datenbank und die aktuell gewählte Tabelle (der Datensatz, über dem exploriert wird)
    - Position: im untersten Süden

Für Funktionen, die nicht direkt mit der Anzeige des Graphen und seinen Informationen zusammenhängen, gibt es nun noch 3 Dialoge, die über die *MenuBar* (nur in Desktop-Applikation) bzw. *ToolBar* aufgerufen werden können:

- *DBDialog:*
    - ist Verantwortlich für die Benutzeraktionen:
        - Herstellen einer Verbindung zu einem Datenbank-System (Verbindungsparameter)
        - Auswahl einer Datenbank (nur von unserer Software erstellte Datenbanken werden angezeigt und akzeptiert, da gleich bei Erstellung die ICD-Table mit den ICD-Codes installiert wird, sowie eine MetaDaten-Tabelle, welche die enthaltenen Datensätze zusammen mit der dazugehörigen Anzahl der untersuchten Patienten enthält)
        - Erstellung einer neuen "Comorbidity"-DB
        - Auswahl eines Datensatzes (einer Tabelle)
            - aber nur zu Beginn, danach wird die Auswahl über das *OptionPanel* auf dem Haupt-Panel übernommen
    - Wichtiger Hinweis:
        - nachdem einmal eine Verbindung hergestellt und eine Datenbank ausgewählt wurde, ist dies im folgenden Programmverlauf nicht mehr möglich! (Diese Aktionen sind "disabled")
        - Der Grund dafür ist, dass wir damit Inkonsistenzen und Fehlerzustände im Programmablauf verhindern wollen

- *DataImportDialog:*
    - zum Importieren von statistischen Daten, im Format der HuDiNe-Daten in die Datenbank:
        - er zeigt die bereits geladenen Datensätze an, weist den Benutzer auf das erwartete Datei-Format hin und bietet mit Textfeldern für Dateiname (der zu importierenden Datei), Tabellenname und die Anzahl untersuchter Patienten die Eingabeschnittstelle zum Daten-Import
        - Importiert werden kann nur, wenn eine Verbindung zu einer "Comorbidity"-Datenbank besteht
        - Die Eingaben aus den Textfeldern werden bevor der Imort startet nochmal validiert (Womit aber noch nicht vollständig ausgeschlossen ist, dass auch fehlerhafte Daten importiert werden können. In einem solchen Fall übernimmt das Datenbanksystem das Weitere, indem die Einträge, die nicht in das gewünschte Format geparst werden können, auch nicht mit importiert werden.)

- *ExportDialog* (ein *JFileChooser* modal zum MainPanel bzw. Frame)
    - zum Exportieren des momentan angezeigten Graphen als Bild und seiner Knoten- und Kantentabelle als CSV-Datei, beide Export-Aktionen werden durch den *DataOptionsController* gehandhandhabt
        - Für den Bild-Export, wird die *ExportDisplayAction* aus dem Paket *prefuse::util::display* verwendet

– Für den Tabellen-Export der Knoten- und Kantentabelle des Graphen gibt es die Klasse *TableExportAction* im paket *comorbidity_viewer::util*

Klassendiagramm dazu in der Datei: "*Comorbidity Viewer view.jpg*"

4.9 Paket: controls

Dieses Paket beinhaltet die externen Controller für die View-Komponenten, welche die Events der Interaktionselemente des Views handeln und auf den "Models" operieren. Diese Models sind im wesentlichen:

- *DataBaseConnector*
- *GraphManager*
- *CVVisualization*
- *CVDisplay*

Aber auch:

- *CVMainPanel*
- *CVFrame*

Die beiden Haupt-Views für den *DataBaseConnector* sind der *DBDialog* und der *DataImportDialog*.

Während der DBController die Interaktionskomponente zwischen *DataBaseConnector* und *DBDialog* darstellt (MVC-Architektur), gibt es für den DataImportDialog keinen externen Controller, er folgt dem MVP-Design.

Beide Views (implementieren *Observer*) enthalten eine Instanz des DataBaseConnectors (erweitert *Observable*), über dessen update-Methoden sie über Änderungen benachrichtigt werden. Der DBController ist zusätzlich auch ein Observer, um die gültigen Zustände und erlaubten Aktionen im DBDialog zu kontrollieren. Er enthält eine Instanz des DataBaseConnectors und zur Kommunikation mit dem View (DBDialog) auch eine Instanz selbigen. Damit dies möglich ist, wird er erst im Konstruktor des DBDialoges instanziiert. Hier sind Daten, Darstellung und Programmlogik streng voneinander getrennt (weshalb sie auch einen hohen Kommunikationsaufwand aufweisen).

Der *DataImportDialog* hingegen ist View und Controller in einem.

Der *SearchPanelController* folgt wieder dem MVC-Konzept und interagiert mit dem *GraphManager* und *DataBaseConnector* als Model und dem *ICDSearchPanel* als View.

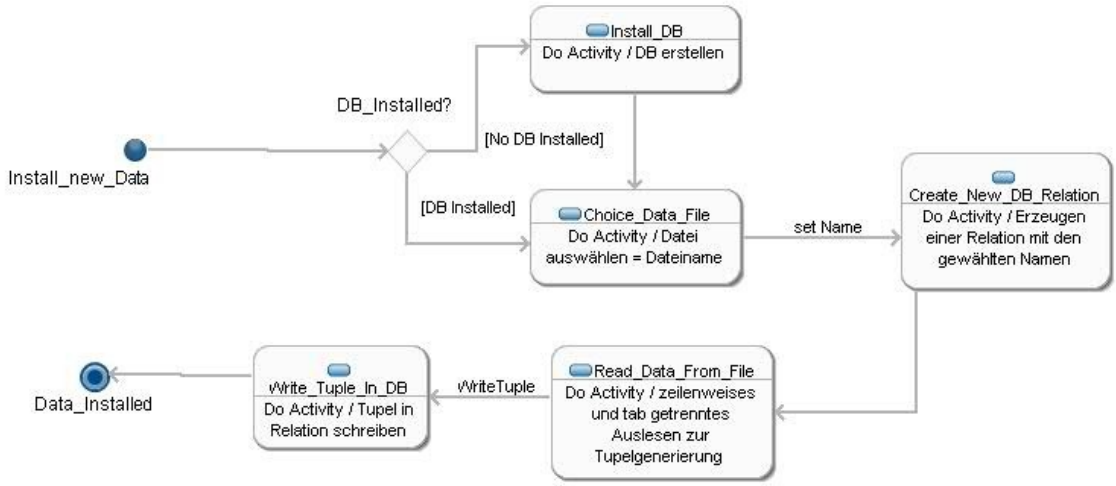
Das wesentliche "Controller-View", über welches die meisten Controller Ereignisse empfangen und verarbeiten ist das OptionPanel mit seinen 4 ToolBars. Es ergibt sich die folgende Struktur:

- *Data-ToolBar:*
  - ➔ *DataOptionsController*
  - ➔ *RelationSwitchController*
- *Layout-Toolbar:*
  - ➔ *LayoutSwitchController*
- *View-ToolBar:*
  - ➔ *CVViewController*
  - ➔ *NodeLabelController*
- *Info-ToolBar:*
  - ➔ *InfoOptionsController*

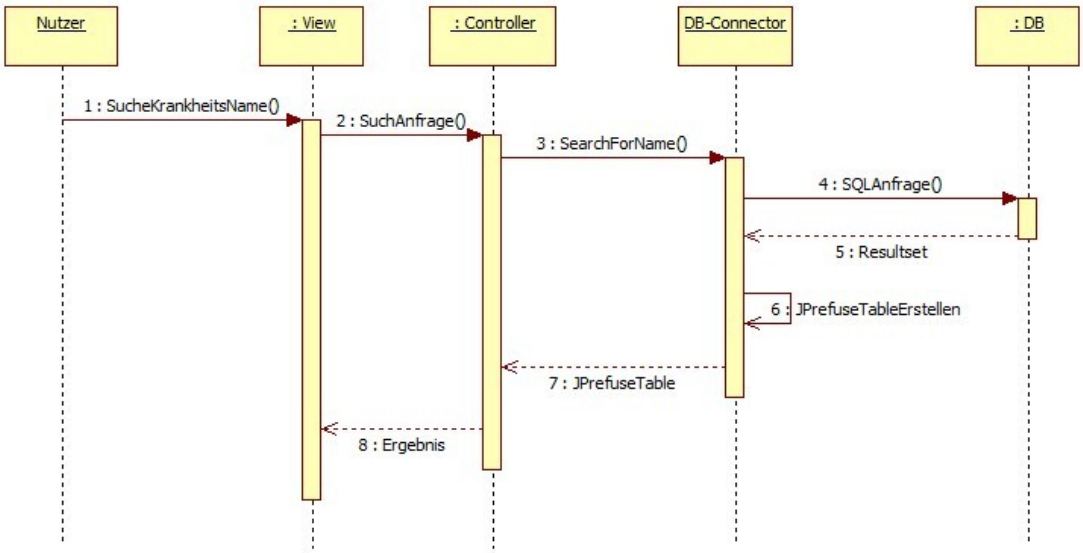
5 Zusammenwirken des Systems

Das Zusammenspiel der Komponenten des Produkts soll anhand von ausgewählten, wichtigen Anwendungsfällen erläutert werden.

5.1 Zustandsdiagramm: Installation neuer Datensätze



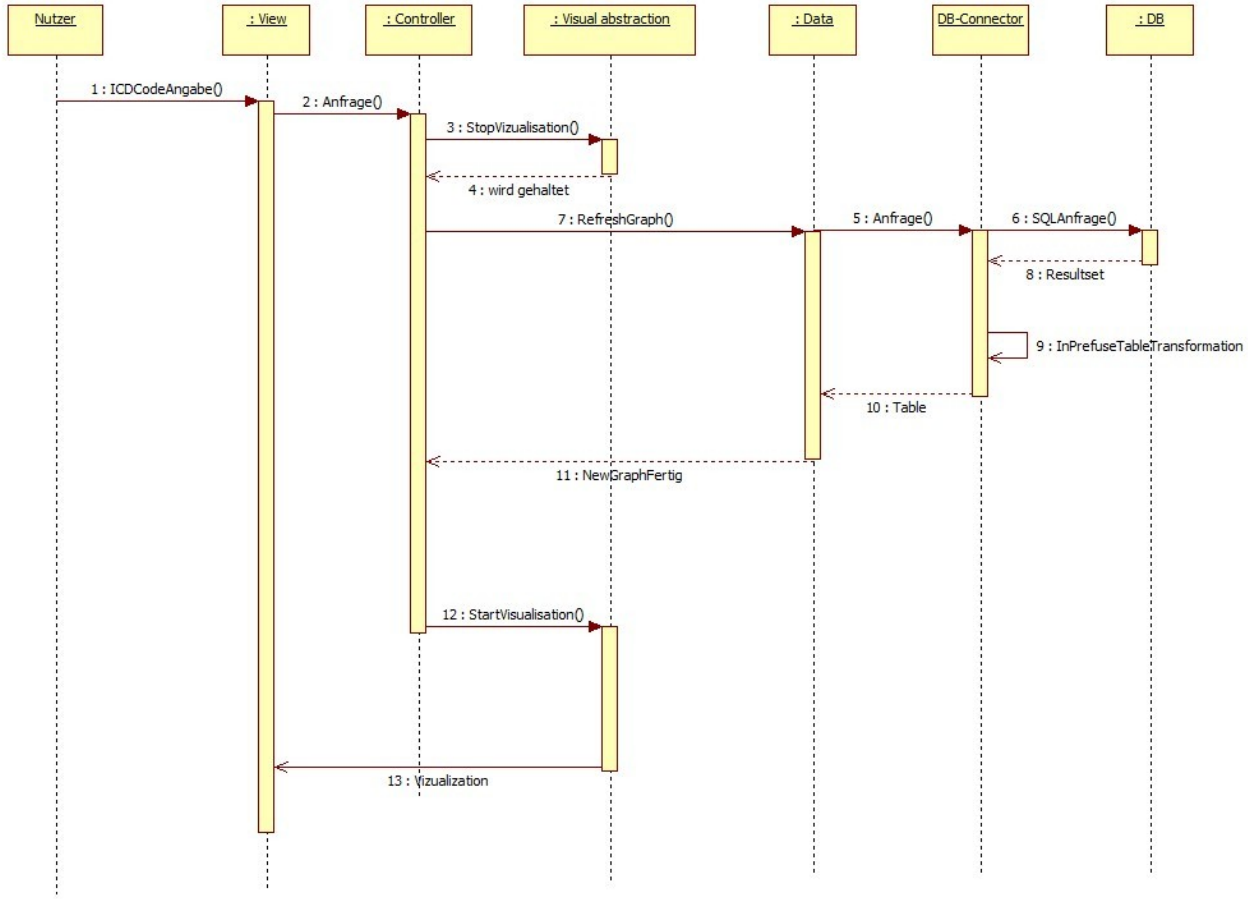
5.2 Sequenzdiagramm: Suche nach Krankheiten in ICD-Code-Tabellen



Beschreibung: Um einen Graphen initialisieren zu können, muss zuerst eine Krankheit per Schlagwort oder ICD-Code ausgewählt werden. Die Suchergebnisse werden in einer Liste angezeigt, aus welcher der Benutzer dann eine Krankheit zum Initalisieren und explorieren des Graphen auswählen kann.

5.3 Sequenzdiagramm: Initialisieren / Explorieren eines Graphen

Ein Sequenzdiagramm für das Initialisieren bzw. das Explorieren eines Graphen bei ausgewählter Krankheit aus dem Suchergebnis bzw. bei zu explorienenden Knoten:



Der Unterschied beim Initialisieren und dem Explorieren besteht darin, dass die Tabellen beim Erstellen eines neuen Graphen geleert werden müssen, dass heißt alle bestehenden Tupel werden sowohl aus der Kanten- als auch aus der Knotentabelle gelöscht. Beim Explorieren werden lediglich Tupel den Tabellen hinuzgefügt.