5 RIVER - Werkzeugunterstützung

Inhaltsangabe

Dieses Kapitel beschreibt das während der Diplomarbeit entstandene Werkzeug *River*[1]. *River* visualisiert vorhandene Ticket-Daten und deren Änderungen aus Change-Request-Systemen und unterstützt so die Analyse der von diesen Daten reflektierten Softwareentwicklungs-Prozesse. Auf eine Übersicht über die Architektur des Werkzeugs folgt ein detaillierte Beschreibung des Entwurfs und der Implementierung der *River*-Komponenten.

5.1 Architektur

Obwohl sich der Zweck und die Aufgabe des Werkzeugs in einem Satz zusammenfassen lässt, wie es soeben in der Einleitung zu diesem Kapitel geschehen ist, setzt sich die komplexe Gesamtaufgabe aus mehreren Teilaufgaben zusammen. Das Werkzeug soll über Basismetriken Daten aus verschiedenen Change-Request-Systemen extrahieren und von ihnen über Datenänderungen informiert werden können. Diese erhobenen Daten sollen in einem gemeinsamen Datenmodell vereinheitlicht und persistiert werden. Basierend auf diesem Modell bereiten Pseudometriken die erhobenen Daten weiter auf. Schließlich werden sie auf einer grafischen Oberfläche zu visualisiert.

Dem Entwurfsgrundsatz “Seperation of concerns” folgend soll die Anwendung aus einzelnen Modulen bestehen, die jeweils genau eine dieser Teilaufgaben realisieren. Die Module besitzen definierte Schnittstellen, über die sie ihre Funktionalität zur Verfügung stellen und Daten austauschen. Die Modularisierung dient dabei dem Erreichen mehrerer Ziele:

Reduzierung der Komplexität der Abhängigkeiten

Die Zerlegung einer komplexen Gesamtaufgabe in kleinere, über definierte Schnittstellen zusammenarbeitende Teilaufgaben führt zu geringer Kopplung zwischen den Modulen sowie hoher Kohäsion innerhalb der Module. Dies erleichtert insbesondere das Verständnis der Anwendung sowie unterstützt die unten angesprochene Wiederverwendbarkeit

[1] Der Name *River* (deutsch: Fluss) leitet sich von der Mengenfluss-Darstellung der Sankey-Diagramme ab, die innerhalb des Werkzeugs eine zentrale Rolle spielen.

einzelner Teile des Werkzeugs. Darüber hinaus verbessert sich die Änderbarkeit: Bei Änderungen an einem Modul bleiben andere Module davon unberührt, sofern die Änderung nicht die Schnittstelle, sondern lediglich die Implementierung des Moduls betrifft.

Die richtige Technologie für die richtige Aufgabe

Die Technologie JSF etwa eignet sich zur Definition von grafischen Oberflächen in Webanwendungen und wird daher hier auch zur Erstellung des GUI genutzt. Andere Module der Anwendung sind aus Eignungsgründen oder aufgrund von Randbedingungen in den Technologien Java EE oder Python umgesetzt. Über die definierten Schnittstellen der Module sind diese verschiedenen Technologien in der Lage, zusammenzuarbeiten und Daten auszutauschen[2].

Wiederverwendbarkeit

Bereits während der Entwicklung des Werkzeugs gab es in der Forschungsgruppe Software Construction Pläne, Teile dieses Werkzeugs im Forschungsprojekt MeDIC-Dashboard wiederzuverwenden. Insbesondere die Datenerfassung und Pseudometrik-Berechnung sollen sich zur Wiederverwendung eignen.

Die eingangs genannten Teilaufgaben bauen jeweils aufeinander auf. Daraus ergibt sich eine noch grobe, technologieunabhängige Schichtenarchitektur (Abbildung 5.1).

Sie wurde von der klassischen 3-Schichten-Referenzarchitektur[Rum10], bestehend aus Präsentations-Schicht, Anwendungslogik-Schicht und Persistierungs-Schicht, abgeleitet. Der Vorteil einer solchen Architektur liegt in der Austauschbarkeit der einzelnen Schichten. Insbesondere im Rahmen einer möglichen Integration in MeDIC-Dashboard ist ein Austausch der Präsentations-Schicht, und ggf. auch eine Adaption der Persistierungs-Schicht zu erwarten.

Die folgenden Überlegungen beschreiben zunächst die Wahl der eingesetzten Technologien und Rahmenwerke und münden in der Erläuterung einer detaillierteren, plattformabhängigen Architektur des Werkzeugs.

Verwendete Technologien

Wie bereits die vorausgehenden Prototypen wurde auch *River* als Webanwendung entwickelt. Ein ganz allgemeingültiger Vorteil einer Webanwendung gegenüber den Alternativen (native Anwendung, Java- oder .NET-Anwendung) war dabei ausschlaggebend: Die Plattformunabhängigkeit aus Sicht des Nutzers/Stakeholders. Um eine moderne Webanwendung zu verwenden, benötigt der Anwender lediglich einen JavaScript-fähigen Webbrowser. Diese Infrastruktur ist auf praktisch jedem verbreiteten internetfähigen Endgerät verfügbar und installiert, unabhängig davon ob es sich um einen PC, einen Mac, ein Tablet oder Smartphone handelt, und ob es unter Windows, OSX, iOS oder Android läuft. Java- und .NET-Anwendungen bieten zwar über ihre auf vielen Systemen verfügbaren Laufzeitumgebungen ebenfalls einen hohen Grad an Plattformunabhän-

[2] Die Auswahl der Technologien wird weiter unten genauer erläutert. Hier ist wichtig, dass verschiedene eingesetzt werden und zusammenarbeiten müssen.

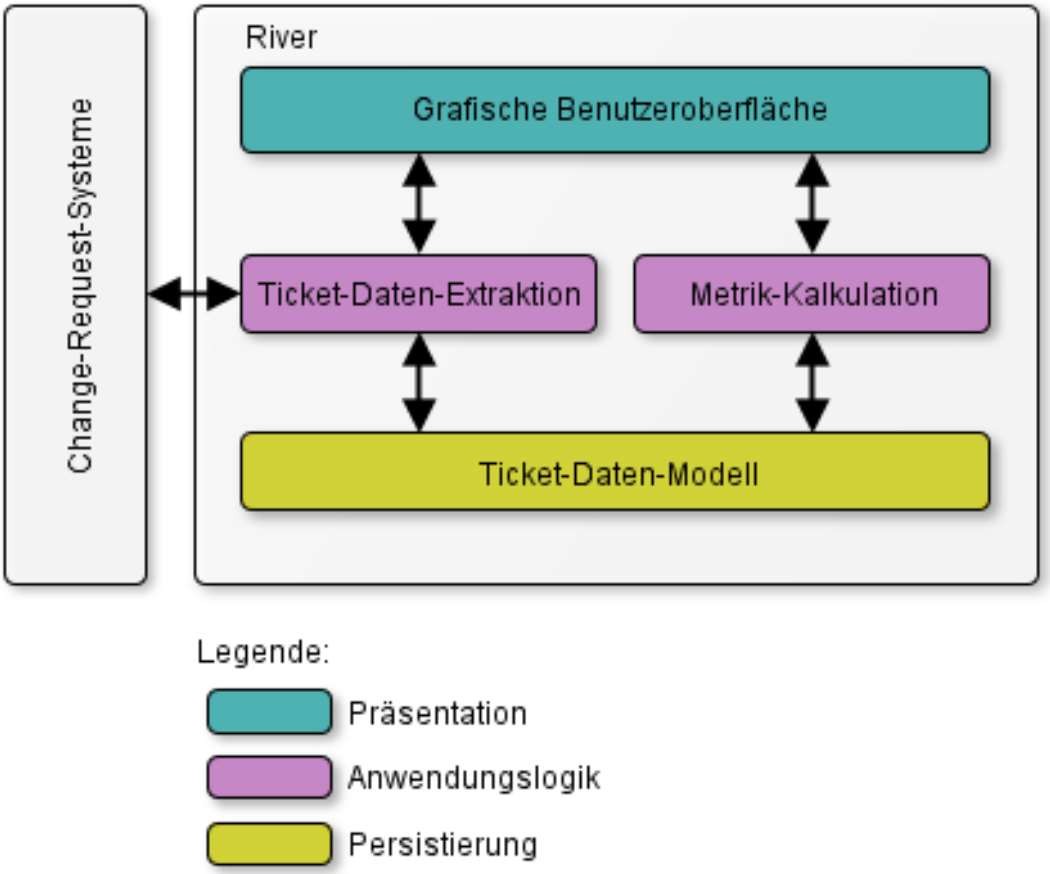

Abbildung 5.1: *River*: Technologieunabhängige Schichten-Architektur

gigkeit. Diese Laufzeitumgebungen sind aber oft nicht vorinstalliert oder auf einzelnen Hardware/Betriebssystem-Kombinationen doch nicht verfügbar.

Es gibt zahlreiche Möglichkeiten und Technologien, um eine Webanwendung zu entwickeln. Eine Stand-Alone Webanwendung etwa benötigt keinen Server und wird vollständig im Webbrowser ausgeführt. Sie besteht aus HTML-Code zur Darstellung der Oberfläche und enthält JavaScript-Code, der die Programmlogik umsetzt. Technologien für komplexere, server-basierte Webanwendungen sind z.B. Ruby on Rails, PHP, .NET oder Java EE. Sie unterscheiden sich in der verwendeten Programmiersprache sowie dem Angebot an Bibliotheken, Komponenten und APIs, die Standardfunktionalität bereitstellen (z.B. Datenstrukturen wie Bäume oder Listen) oder die Verwendung weiterer Technologien wie z.B. XMLRPC unterstützen.

Zur Entwicklung von *River* wurde (mit Ausnahme des in Python geschriebenen Trac-Plugins zur Datenaktualisierung) die Java Platform, Enterprise Edition in Version 6 (Java EE 6) verwendet. Ausschlaggebend für die Entscheidung für die Java EE Technologie waren folgende drei Gründe:

Gute Eignung zur Umsetzung der Anforderungen

Java EE unterstützt und erleichtert über diverse APIs unmittelbar die Umsetzung einiger Anforderungen an das Werkzeug. Beispielsweise dient die Java Persistence API (JPA) der Speicherung von Daten innerhalb der Anwendung, wie sie die Persistierungs-Schicht durchführt. Ebenfalls von der Persistierungs-Schicht wird die Java Transaction API (JTA) genutzt, um die konsistente Extraktion der Daten sicherzustellen. Der eigentliche Zugriff aus dem Werkzeug auf Change-Request-Systeme gelingt über eine XMLRPC API. Die Werkzeugoberfläche schließlich wird mit Hilfe von Java Server Faces (JSF) definiert.

Mögliche Integration in MeDIC-Dashboard

Wie oben bereits angeführt, sollen einzelne Komponenten des Werkzeugs ggf. in MeDIC-Dashboard wiederverwendet werden. MeDIC-Dashboard ist ebenfalls in der Technologie Java EE implementiert. Die Wahl derselben Technologie erleichtert eine zukünftige Integration.

Erfahrung mit der Technologie

Der Diplomand besaß aufgrund einer der Diplomarbeit vorgelagerten HIWI-Tätigkeit bereits Erfahrungen mit Java EE. Dies ermöglichte aufgrund einer kürzeren Einarbeitungszeit in die Technologie eine stärkere Fokussierung auf die fachliche Problemstellung während der Diplomarbeit.

Technologieabhängige Architektur

Anhand der Entscheidungen, welche Technologien das Werkzeug einsetzen soll, wurde das technologieunabhängige Architekturdiagramm (Abbildung 5.1) zu einem technologieabhängigen (Abbildung 5.2) verfeinert. Es stellt dabei mehrerer Aspekte der Architektur in einer Ansicht dar. Die Einfärbung der Module zeigt ihre Schicht-Zugehörigkeit innerhalb der 3-Schichten-Architektur. Die Pfeile visualisieren die Kommunikation zwischen den Modulen. Ein dünner Pfeil steht für modulübergreifende Methodenaufrufe und visualisiert so den Kontrollfluss. Ein gepunkteter Pfeil symbolisiert den Datenfluss innerhalb und außerhalb der Anwendung. Hierbei handelt es sich um Ticket- und Metrik-Daten.

Der folgende Überblick skizziert die Rollen und die Zusammenarbeit der einzelnen Komponenten, die später in diesem Kapitel in dedizierten Abschnitten genauer erläutert werden: Der Nutzer kann über das GUI (das dazu über die *Fassade* und den *Datenquellen-Controller* das Modul *initiale Datenerfassung* anspricht) eine neue bzw. bereits verwendete Datenquelle angeben und den initialen Datenimport bzw. erneuten Datenimport aus dieser Quelle anstoßen. Dazu extrahiert das Modul *initiale Datenerfassung* Ticket-Daten aus der Datenquelle entweder über den XMLRPC-Dienst der Datenquelle oder aus einer aus dem Change-Request-System exportierten Ticket-Daten-Datei. Das Modul normalisiert diese Ticket-Daten und leitet sie über den JMS Nachrichtenbus an die *Ticket-Daten-Verarbeitung* weiter. Dort werden sie weiter aufbereitet und anschließend in der Ticket-Daten-Persistierung gespeichert. Wieder ausgehend von

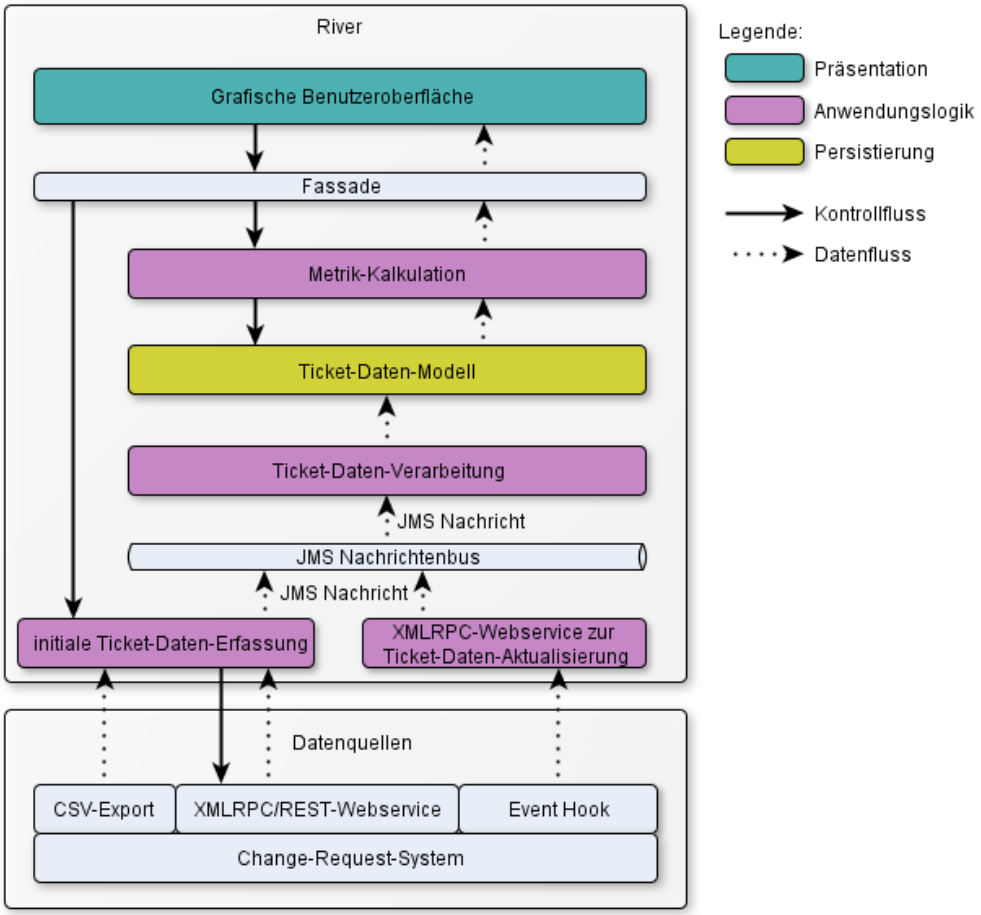

Abbildung 5.2: *River*: Technologieabhängige Architektur

einer Nutzerinteraktion fragt das GUI-Modul berechnete Metrik-Daten von den Metrik-Kalkulatoren ab und visualisiert sie anschließend auf seiner Oberfläche. Die Metrik-Kalkulatoren greifen dazu auf die zuvor in der Ticket-Daten-Persistierung gespeicherten Ticket-Daten zurück. Um im laufenden Betrieb die Ticket-Daten der Datenquelle und die in der Ticket-Daten-Persistierung des Werkzeugs gespeicherten Ticket-Daten konsistent zu halten, benachrichtigt die Datenquelle bei einer Datenänderung das Modul *Datenaktualisierung*, das ähnlich wie die *initiale Datenerfassung* die erhaltenen Ticket-Daten normalisiert und über den Nachrichtenbus weiterleitet.

5.2 Grafische Benutzeroberfläche (GUI)

Das GUI dient als Schnittstelle zwischen dem Anwender und dem Werkzeug. Es bietet dem Anwender zwei grundsätzliche Funktionalitäten an, die die Webanwendung über zwei verschiedene Webseiten zur Verfügung stellt. Diese Webseiten sind über eine oben im Werkzeug angezeigte Navigationsleiste erreichbar (siehe Abbildung 5.3).

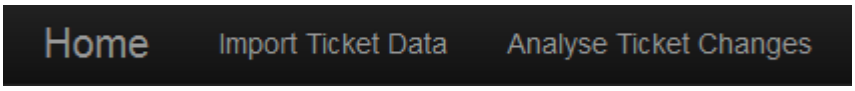

Abbildung 5.3: Navigationsleiste in *River*

Import Ticket Data

Über die Seite “Import Ticket Data” stößt der Anwender den initialen oder erneuten Ticket-Daten-Import aus einer Datenquelle an. Die dazu nötigen Informationen gibt er über ein Formular ein.

Analyse Ticket Changes

Hier analysiert der Anwender die importierten Ticket-Daten, um Rückschlüsse auf den durch sie reflektierten Softwareentwicklungs-Prozess zu ziehen. Er wählt dazu über GUI-Eingabeelemente die gewünschte Datenquelle sowie die zu analysierende Ticket-Eigenschaft aus und spezifiziert optional einen Filter. Das Werkzeug visualisiert daraufhin die Änderungshistorie der Tickets bezüglich der gewählten Ticketeigenschaft in einem Sankey-Diagramm. In diesem Sankey-Diagramm kann der Benutzer Diagramm-Elemente (Knoten und Transitionen) anwählen und erhält über rechts angeordnete Detailfenster weitere Kontext-Informationen zu der durch das Diagramm-Element repräsentierten Ticketgruppe.

Implementierung

Das GUI wurde in dem Java EE Framework-Standard zur Entwicklung grafischer Oberflächen *Java Server Faces (JSF)* (siehe Abschnitt 2.5) realisiert und verwendet die JSF-Komponenten-Bibliothek Primefaces.

Quelltext 5.1 und Quelltext 5.2 illustrieren die Definition des GUI-Elements (eine DropDown-Liste) zur Auswahl der Datenquelle auf der Seite “Analyse Ticket Changes”

der Anwendung. Das JSF-Tag “<f:selectItems value="#sankeyBean.dataSources"/>” definiert die Listeneinträge der DropDown-Liste und greift dazu auf den Getter *getDataSources()* zurück, der wiederum die Liste über einen Methodenaufruf an die Fassade zusammenstellt. Wählt der Anwender eine Datenquelle aus, wird zunächst das Datenmodell über den Setter *setDataSource* aktualisiert und anschließend die Controller-Methode *dataSourceChanged*[3] aufgerufen. Sie setzt im Datenmodell sämtliche sonstigen gemachten Benutzereingaben (wie die Auswahl der Ticketeigenschaft oder des Filters) zurück, da sie nach einer Änderung der Datenquelle nicht mehr gültig sind. Schließlich wird über die Deklarationen “oncomplete=...” und “update=...” definiert, welche Teile der View aktualisiert werden müssen, d.h. potentiell geänderte Daten des Datenmodells anzeigen. Üblicherweise genügt dazu eine “update=...”-Angabe. In diesem Fall handelt es sich bei dem Sankey-Diagramm jedoch um keine JSF-Komponente, weshalb eine JavaScript-Funktion das Zurücksetzen des Diagramms übernimmt.

```
1  <h:form id="selectDatasourceForm">
2    <p:selectOneMenu id="selectDatasource" style="width:100%;" value="
       #{sankeyBean.dataSource}">
3      <p:ajax listener="#{sankeyBean.dataSourceChanged}"
4        oncomplete="clearSankey();"
5        update=":selectTicketPropertyForm:selectTicketProperty :filters
             :ticketCounter :selectedTicketsForm:selectedTicketsTable
            :selectHistogramPropertyForm:selectHistogramProperty
            :histogram" />
6      <f:selectItem itemLabel="Please select" itemValue="" />
7      <f:selectItems value="#{sankeyBean.dataSources}" />
8    </p:selectOneMenu>
9  </h:form>
```

Quelltext 5.1: Auszug aus sankey.xhtml

```
1  [...]
2
3  @ManagedBean
4  @ViewScoped
5  public class SankeyBean extends BeanBase implements Serializable {
6
7          @EJB
8          private GplcrsFacadeLocal facade;
9
10         private String dataSource;
11
12         public String getDataSource() {
13                 return dataSource;
14         }
15
16         public void setDataSource(String dataSource) {
17                 this.dataSource = dataSource;
18         }
19
```

[3] Die in dataSourceChanged() genutzten Methoden (clearTicketPropertySelection() usw.) sind ebenfalls in SankeyBean.java definiert, hier aber aus Gründen der Übersichtlichkeit des Listings weggelassen.

```
20          public Map<String, String> getDataSources() {
21                  Map<String, String> dataSourcesMap = new TreeMap<>();
22                  List<String> dataSourcesList = facade.
                        getDataSourceIdentifiers("");
23                  for (String dataSourceIdentifier : dataSourcesList) {
24                          dataSourcesMap.put(dataSourceIdentifier,
                                dataSourceIdentifier);
25                  }
26                  return dataSourcesMap;
27          }
28
29          public void dataSourceChanged() {
30                  clearTicketPropertySelection();
31                  clearSankey();
32                  clearSelectedTickets();
33                  clearFilters();
34                  clearHistogramProperties();
35                  clearHistogramModel();
36          }
37
38          [...]
39
40  }
```

Quelltext 5.2: Auszug aus SankeyBean.java

Seite: Import Ticket Data

Die Seite "Import Ticket Data" ermöglicht dem Anwender, Ticket-Daten aus einer Datenquelle in das Werkzeug zu importieren. Das Werkzeug erlaubt grundsätzlich gemäß Anforderung (Q4) die Anbindung an beliebige Datenquellen über Datenquellen-Adapter. Dazu sind die für den Import benötigten Informationen (wie etwa Zugangsdaten zu einem Change-Request-System) festzulegen, die GUI entsprechend anzupassen, um eine Eingabe dieser Informationen zu ermöglichen und schließlich eine Java-Klasse, die über eine Implementierung des *DataSource*-Interface des Werkzeugs den Zugriff auf die Datenquelle umsetzt.

Im Rahmen dieser Diplomarbeit wurden exemplarisch drei Datenquellen-Adapter für die Change-Request-Systeme Trac, Redmine und ClearQuest entwickelt. Bei der Auswahl dieser drei Change-Request-Systeme war ausschlaggebend, dass Projektleiter und Softwareprozess-Manager aus Industrie- und Universitäts-Softwareentwicklungs-Projekten, die diese Systeme nutzen, eine Evaluation des Werkzeugs an ihren Projekten ermöglichten. Trac wird aufgrund seiner Integration in SSELab in vielen Softwareentwicklungs-Projekten der Fachgruppe Informatik an der RWTH Aachen verwendet. Redmine und ClearQuest werden von je einem Kooperationspartner dieser Diplomarbeit eingesetzt[4]. Die drei implementierten Datenquellen-Adapter lassen sich in zwei Kategorien einteilen.

[4]Die eigentlichen Import-Mechanismen der drei Datenquellen-Adapter werden in Abschnitt 5.5 beschrieben. Dieser Abschnitt beschreibt die Aspekte, die das GUI betreffen

Import über eine API des Change-Request-Systems

In diese Kategorie fallen die Datenquellen-Adapter für Trac und Redmine. Beide Change-Request-Systeme bieten eine XMLRPC-Schnittstelle an, über die das Werkzeug Ticket-Daten und Ticket-Historien abfragt. Dazu wird im Change-Request-System ein Benutzer-Account benötigt. Der *River*-Anwender gibt die URL des Change-Request-Systems, sowie Benutzername und Kennwort des Trac-Benutzer-Accounts (siehe Abbildung 5.4) bzw. den API-Key des Redmine-Benutzer-Accounts an und kann daraufhin den Import starten. Im Werkzeug vorhandene Ticket-Daten und Ticket-Historien zu einem Trac- oder Redmine-Change-Request-System werden bei einem erneuten Import aus dem selben System zunächst gelöscht.

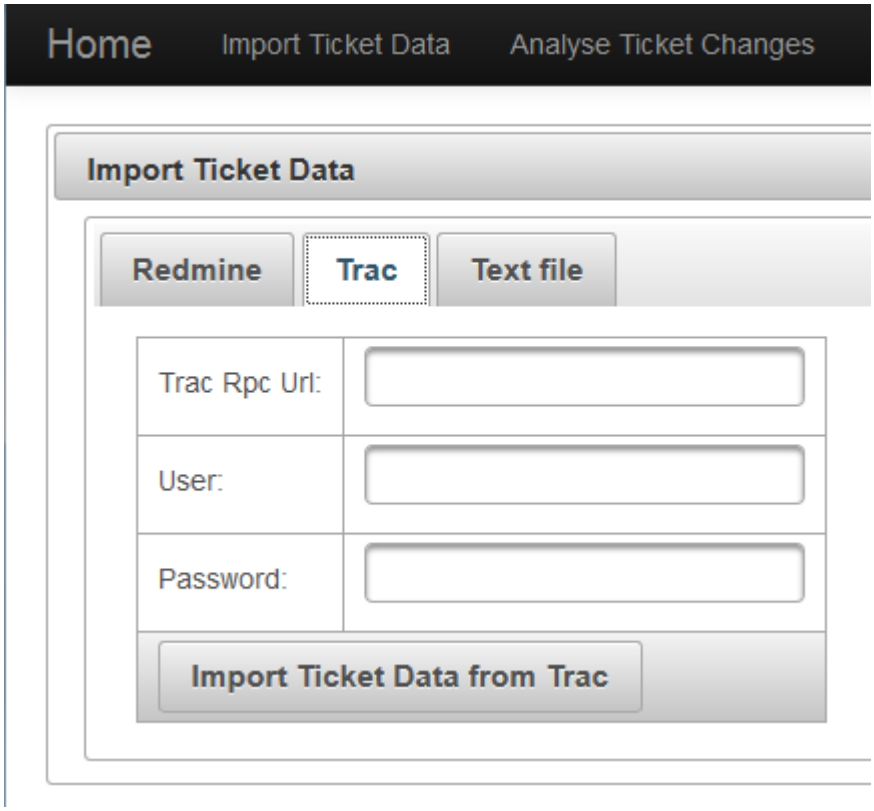

Abbildung 5.4: Import aus einem Trac-System

Import aus einer Datei

Der ClearQuest-Kooperationspartner hat statt eines API-Zugriffs mehrere Schnappschüsse mit Ticket-Daten des ClearQuest-Systems zu verschiedenen Zeitpunkten in Form von Comma-Seperated-Value (CSV)-Dateien zu Verfügung gestellt. Diese Dateien enthielten den Zustand der Tickets zum jeweiligen Schnappschuss-Erstellungs-Zeitpunkt. Nicht enthalten waren Ticket-Historien. Diese Schnappschuss-Dateien werden sukzessiv über die Datei-Import-Funktion des Werkzeugs importiert. Dazu gibt der *River*-Anwender den Schnappschuss-Dateinamen, die Zeichenkodierung der Datei (z.B. UTF-8), das Spalten-Trennzeichen sowie die Titel der Spalten für Ticket-Id und Ticket-Thema

an (siehe Abbildung 5.5). Da exakte Ticket-Historien fehlen und dem Werkzeug lediglich die Entwicklung der Tickets von Schnappschuss zu Schnappschuss bekannt gemacht wird, sind Analysen, die auf Ticketdaten aus dieser Importvariante basieren, ungenauer.

Seite: Analyse Ticket Changes

Nach dem Import aus einer Datenquelle visualisiert das Werkzeug auf der Seite "Analyse Ticket Changes" die Ticket-Daten und ihre Historie. Dazu wählt der Benutzer zunächst den zu visualisierenden Datensatz aus (z.B. den gerade zuvor importierten). Anschließend bestimmt er die Ticketeigenschaft, deren Änderungen im Sankey-Diagramm dargestellt werden sollen. Im Gegensatz zu den vorangegangenen Prototypen, die ausschließlich die Ticketeigenschaft "Ticket-Status" betrachteten und gemäß Anforderung (FA3) kann der Anwender mit Hilfe des Werkzeugs die Veränderung aller Ticketeigenschaften untersuchen. Ihm stehen hierbei alle Ticketeigenschaften zur Verfügung, die im zugrundeliegenden Ticket-Datensatz bei mindestens einem Ticket eine Veränderung aufweisen.

Filter

Optional kann der Benutzer Filterregeln festlegen, um eine Untermenge der Tickets eines Change-Request-Systems zu analysieren. Die Filterregeln ähneln den Filterregeln der Ticket-Abfrage-Mechanismen der Change-Request-Systeme Trac und Redmine. Eine Filterregel besteht aus einer Ticketeigenschaft, einem Vergleichsoperator und einem Vergleichswert. Die Ticketeigenschaft wählt der Benutzer aus einer Liste aller Ticketeigenschaften der Tickets der Datenquelle aus. Als Vergleichsoperatoren stehen "ist" und "ist nicht" zur Verfügung. Bei der Eingabe des Vergleichswerts wird der Benutzer durch eine Auto-Vervollständigung unterstützt. Die angebotenen Vorschläge sind Ticketeigenschafts-Werte, die in mindestens einem Ticket aus der gewählten Datenquelle bei der im Filter gewählten Ticketeigenschaft vorkommen und die vom Benutzer bereits eingegebene Zeichenkette enthalten. Die Auto-Vervollständigung beschleunigt so die Eingabe des Vergleichswerts und hilft, Eingabe-Fehler zu vermeiden.

Der Anwender kann mehrere Filterregeln angeben. Diese werden logisch mit dem Operator *UND* verknüpft. Weiter beziehen sich alle Filterregeln auf den aktuellen Zustand der Tickets der gewählten Datenquelle, nicht etwa auf frühere Zustände aus der Historie der Tickets. Im Sankey-Diagramm werden nur diejenigen Tickets und Änderungen an deren Ticketeigenschaften visualisiert, für die alle angegebenen Filterregeln zutreffen.

Ein Beispiel: Der Benutzer möchte Ticket-Daten zu einem bestimmten Projekt aus einem Change-Request-System analysieren, das von mehreren Projekten genutzt wird. Die Tickets besitzen eine Eigenschaft "Projekt", in der der Name des Projekts eingetragen ist, zu dem dieses Ticket gehört. Der Benutzer erstellt nun eine Filterregel für die Ticketeigenschaft "Projekt", wählt den Vergleichsoperator "ist" und trägt den Namen des zu untersuchenden Projekts als Vergleichswert ein. Weiter möchte der Benutzer lediglich wichtige Tickets des Projekts analysieren. Er definiert dazu z.B. zwei weitere Filterregeln: "Priorität" "ist nicht" "Niedrig" und "Priorität" "ist nicht" "Normal".

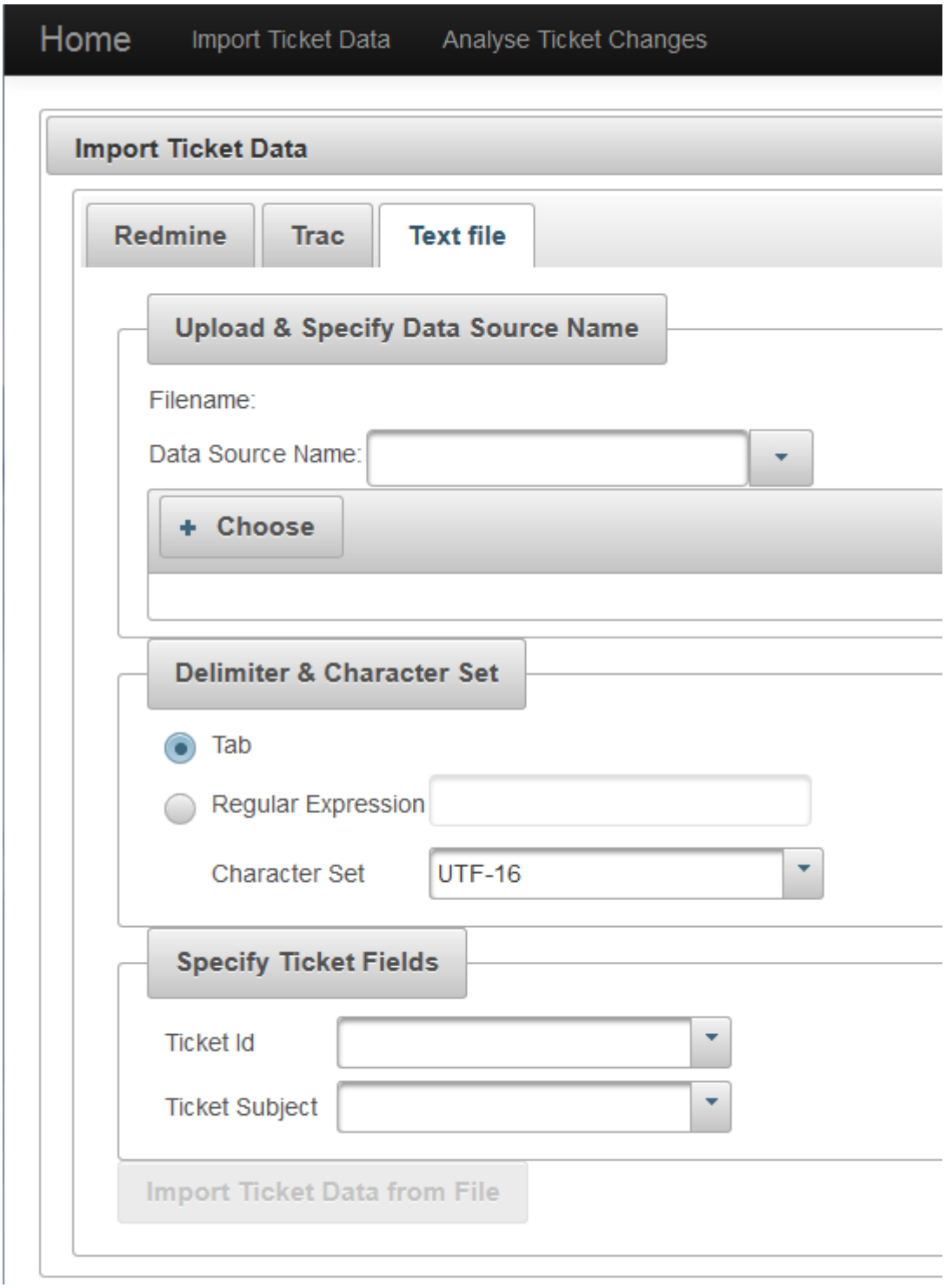

Abbildung 5.5: Import aus einer exportierten Ticket-Daten-Datei

Visualisierung der Ticketeigenschafts-Änderungen

Das Kernstück des Werkzeugs bei der Analyse der Ticket-Daten eines Change-Request-Systems ist ein Sankey-Diagramm, das auf der zuvor erprobten Variante 2 des zweiten Prototyps (siehe Abschnitt 4.7) basiert. Einige Aspekte des Sankey-Diagramms wurden gegenüber seiner Ausprägung im zweiten Prototyp allerdings verändert:

"Hinein fließende" Pfeile: Im Prototyp wurden im Betrachtungs-Zeitraum neu erstellte Tickets, deren Status sich anschließend änderte, durch von oben in das Diagramm "hinein fließende" Pfeile symbolisiert. Sie waren so von den Tickets zu unterscheiden, die bereits vor dem Betrachtungs-Zeitraum erstellt wurden, und deren Status sich im Zeitraum erstmals änderte. Diese "hinein fließenden" Pfeile wurden in den Korridor der Tickets, die im Status "new" gestartet sind, eingebunden, da sie sich lediglich im Zeitpunkt ihrer Erstellung unterschieden, nicht aber in ihrer Status-Historie. Aufgrund der Generalisierung der dargestellten Ticketeigenschaft (von der festen Vorgabe "Status" im Prototypen zu beliebigen Ticketeigenschaften im Werkzeug *River*) gibt es nicht mehr den einen besonderen Korridor "new" in den neue Tickets "hineinfließen". Vielmehr müssten neue Tickets an den Korridor anbinden, dessen Startknoten den gleichen Wert in der im Diagramm dargestellten Ticketeigenschaft repräsentiert, wie das neue Ticket in dieser Eigenschaft bei seiner Erstellung besaß. Weiter ist das Ziel des "hinein fließenden" Pfeils derjenige Knoten in der zweiten Spalte, der für den Wert der Diagramm-Ticketeigenschaft steht, auf den sich die Eigenschaft des neuen Tickets bei der ersten Modifikation ändert. Statt einiger Pfeile oben im Diagramm wie im Prototypen wären bei der Umsetzung dieses Ansatzes viele "hinein fließende" Pfeile im gesamten linken Bereich des Diagramms zu erwarten. Dem geringen Informationsgewinn (Wurde ein Ticket vor oder nach Beginn des Betrachtungs-Zeitraums erstellt?) steht eine starke Verschlechterung der Übersichtlichkeit und Verständlichkeit gegenüber. Daher wurde im Werkzeug auf "hinein fließende" Pfeile verzichtet. Die durch sie repräsentierten Tickets werden stattdessen dem Startknoten des entsprechenden Korridors zugeschlagen.

Entfernung der konsolidierenden Transitionen: Im Prototypen gab es zum einen Transitionen, die echte Änderungen im Status repräsentierten. Zum anderen existierte die letzte Transition jedes Pfades lediglich dazu, die Tickets aus allen Korridoren gemäß ihres Zustands am Ende des Betrachtungs-Zeitraums konsolidiert in der End-Spalte rechts im Diagramm darzustellen. Um diese Zweideutigkeit der Transitionen zu beheben, wurde diese letzte Transition und die konsolidierte Darstellung der Ticket-Verteilung am Ende des Betrachtungs-Zeitraums gemäß der gewählten Ticketeigenschaft entfernt. Nachteilig wirkt sich die nun fehlende Symmetrie der linken und rechten Spalte des Diagramms aus. Diese Entwurfs-Entscheidung wurde in der Evaluation (Kapitel 6) diskutiert.

Auswählbare Knoten und Transitionen: Die Elemente des Sankey-Diagramms wurden selektierbar gemacht. Jeder Knoten und jede Transition im Sankey-Diagramm repräsentiert eine Gruppe von Tickets. Durch die Selektion eines Diagramm-Elements wird diese Gruppe ausgewählt. In den rechts des Sankey-Diagramms angeordneten

Detailfenstern werden weitere Informationen zu den ausgewählten Tickets angezeigt.

Detailfenster: Ausgewählte Tickets

In diesem Detailfenster wird eine Liste der im Sankey-Diagramm ausgewählten Ticketgruppe dargestellt. Die Liste zeigt zu jedem ausgewählten Ticket die Ticket-Id und das Ticket-Thema an. Ein Listeneintrag ist gleichzeitig ein Link zur URL des Tickets im Change-Request-System, falls eine solche URL vorhanden ist[5]. Über diesen Link kann der Anwender weitere Informationen über die ausgewählten Tickets direkt aus dem Change-Request-System abrufen.

Detailfenster: Histogramm über weitere Ticketeigenschaft

Das zweite Detailfenster unterhalb der Ticket-Liste stellt die ausgewählten Tickets in den Kontext einer weiteren, durch den Benutzer auswählbaren Ticketeigenschaft. Die Verteilung der im Sankey-Diagramm ausgewählten Tickets wird gemäß dieser zweiten ausgewählten Eigenschaft in einem Histogramm dargestellt. Der Benutzer erhält so weitere Informationen, die ihn bei der Analyse des Prozesses unterstützen.

Ein Beispiel (siehe Abbildung 5.6): Der Benutzer wählt den Ticket-Status als im Sankey-Diagramm zu visualisierende Eigenschaft aus, um Abweichungen vom gewünschten Workflow zu identifizieren. Anschließend definiert er den Filter “Status” “ist nicht” “Geschlossen”, da er die derzeit aktiven Tickets untersuchen möchte. Er selektiert eine Transition, die Teil eines nicht gewünschten Ticket-Status-Verlaufs ist. Als zweite, im Histogramm darzustellende Ticketeigenschaft wählt er die Ticket-Priorität aus. Diese Visualisierung zeigt, ob es sich bei den ausgewählten Tickets vorwiegend um Tickets mit hoher oder mit niedriger Priorität handelt. Sie kann so zur Entscheidung des Benutzers, ob und wenn ja, welche Maßnahmen zur Korrektur des Prozesses erforderlich sind, beitragen.

5.3 Daten-Modell und -Persistierung

Das Ticket-Datenmodell dient der vereinheitlichten Repräsentation und der Persistierung der aus den Change-Request-Systemen importierten Ticket-Daten und Ticket-Historien. Bei einem Import-Vorgang aus einer Datenquelle werden die Ticket-Daten normalisiert und im Ticket-Datenmodell gespeichert. Zur Visualisierung greift das Werkzeug dann auf die im Modell gespeicherten Ticket-Daten zurück. Das Modell entkoppelt also durch seine einheitliche Daten-Darstellung die zur Visualisierung genutzte Metrik-Kalkulation von den Change-Request-Systemen und insbesondere von den dort genutzten, Change-Request-System-spezifischen Ticket-Datenformaten. Das Modell besteht aus Ticket-Daten und Ticket-Historien:

[5] Dies ist bei Trac- und Redmine-Datenquellen der Fall. Bei einer Datenquelle, die auf einem Ticket-Daten-Datei-Import basiert, entfällt der Link.

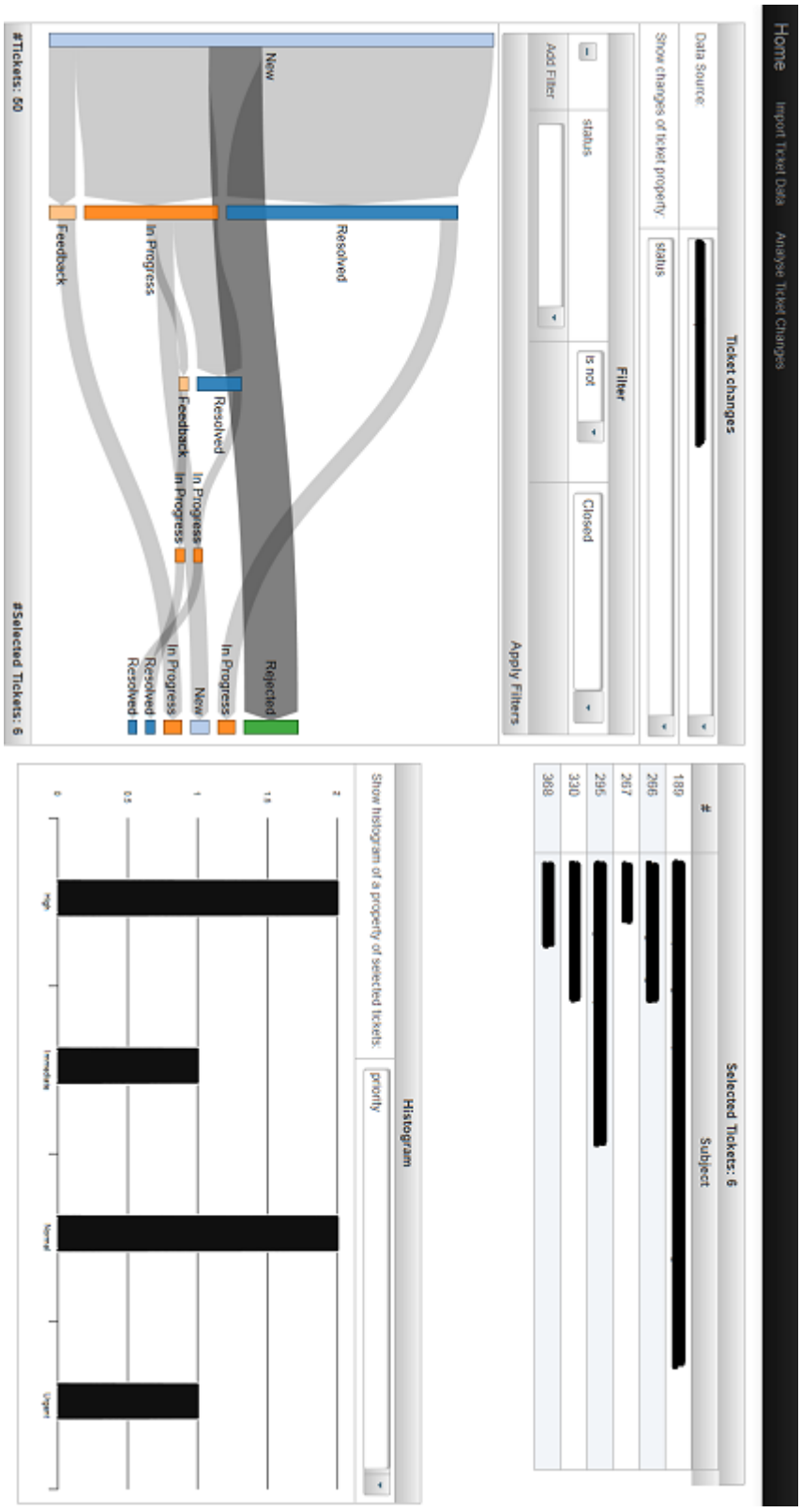

Abbildung 5.6: Analyse mit *River*

Ticket-Daten

Die Ticket-Daten repräsentieren die aktuellen Zustände der Tickets zum Import-Zeitpunkt bzw. nach Datenaktualisierungen. Jedes Ticket wird im Daten-Modell durch seine Eigenschaften und Metadaten charakterisiert. Einige dieser Eigenschaften haben im Daten-Modell eine spezifische Semantik. Der *dataSourceIdentifier* ist eine Kennung der Datenquelle aus der das Ticket stammt. Er ist somit keine eigentliche Eigenschaft des Tickets, sondern ein Metadatum und dient dazu, Tickets aus mehreren Datenquellen in einem Modell zu verwalten. Die *ticketId* identifiziert ein Ticket. Sie ist innerhalb einer Datenquelle eindeutig. Gemeinsam mit dem *dataSourceIdentifier* stellt die *ticketId* eine eindeutige Kennung des Tickets im Modell dar. Das *subject* und die *url* sind weitere, im Modell gesondert notierte Eigenschaften des Tickets. Sie werden in der Visualisierung im "Detailfenster: Ausgewählte Tickets" angezeigt, wobei die *url* zur Generierung eines Links optional ist. Daraus ergeben sich die Minimalanforderungen an die aus einer Datenquelle zu extrahierenden Ticket-Daten. Sie müssen eine Ticket-Id und ein Ticket-Thema enthalten. Alle weiteren Eigenschaften sind optional. Sie werden einem Verzeichnis abgelegt, das den Eigenschaftsnamen auf den Eigenschaftswert eines Tickets abbildet. Dieses Verzeichnis ist durch die generische Java-Datenstruktur *HashMap<String,String>* realisiert. Durch die Verwendung eines Verzeichnisses zur Speicherung der Ticketeigenschaften besitzt das Modell die Flexibilität, Tickets verschiedener Change-Request-Systeme aufzunehmen, die nahezu beliebige Ticketeigenschaften enthalten, solange sich die Eigenschaftswerte sinnvoll im Java-Datentyp *String* darstellen lassen. Das beinhaltet textuelle Ticketeigenschaften wie das Ticket-Thema, die Ticket-Beschreibung oder den Ticket-Autor, ebenso wie numerische Ticketeigenschaften wie eine Programm-Versionsnummer oder den abgeschätzten Arbeitsaufwand zur Lösung des Tickets. Viele Change-Request-Systeme verwenden allerdings auch nicht-textuelle Ticketeigenschaften. Sie erlauben beispielsweise das Anhängen einer Datei an ein Ticket. Diese Eigenschaften werden in diesem Modell nicht berücksichtigt und Änderungen an diesen Eigenschaften können mit *River* nicht analysiert werden.

Ticket-Historie

Die Ticket-Historie speichert Informationen über die Werte, die die Ticketeigenschaften der in den Ticket-Daten gespeicherten Tickets zu früheren Zeitpunkten besaßen. Dazu enthält eine Ticket-Änderung zur eindeutigen Zuordnung zu einem Ticket den *dataSourceIdentifier* und die *ticketId*. Des Weiteren sind der Änderungszeitpunkt (*changeTime*), der Name der geänderten Ticketeigenschaft (*propertyName*), sowie der alte (*oldValue*) und der neue (*newValue*) Wert der Ticketeigenschaft in der Ticket-Änderung enthalten. Ausgehend vom aktuellen Zustand eines Tickets lässt sich anhand dieser Informationen der frühere Zustand eines Tickets rekonstruieren, indem iterierend über die zugehörigen Ticket-Änderungen, beginnend mit der jüngsten, der aktuelle Wert der jeweils betroffenen Ticketeigenschaft durch den *oldValue* ersetzt wird.

Technisch wurde das Daten-Modell durch zwei Entitäten realisiert, die über die Java Persistence API (JPA) (siehe Abschnitt 2.5) in einer zugrundeliegenden Datenbank gespeichert werden. Die folgenden Listings zeigen die beiden Entitäten *Ticket* und *TicketChange*. Anhang A.4 enthält das SQL-Skript zur Erzeugung des

Datenbank-Schemas.

```
1  [...]
2  @Entity
3  public class Ticket implements Comparable<Ticket> {
4  [...]
5          private String dataSourceIdentifier;
6          private Long ticketId;
7          private String subject;
8          private String url;
9
10         @ElementCollection
11         @MapKeyColumn(name = "propertyName", table = "ticket_property
               ")
12         @Column(name = "propertyValue", table = "ticket_property")
13         @CollectionTable(name = "ticket_property", joinColumns =
               @JoinColumn(name = "ticket_id"))
14         private Map<String, String> properties = new HashMap<String,
               String>();
15 [...]
16 }
```

Quelltext 5.3: Entität Ticket (Auszug aus Ticket.java)

```
1  [...]
2  @Entity
3  public class TicketChange {
4  [...]
5          private String dataSourceIdentifier;
6          private Long ticketId;
7          private Long changeTime;
8          private String propertyName;
9          private String oldValue;
10         private String newValue;
11 [...]
12 }
```

Quelltext 5.4: Entität TicketChange (Auszug aus TicketChange.java)

5.4 Metrik Kalkulation

Das zuvor beschriebene Datenmodell stellt die Grundlage zur Berechnung der Pseudo-Metriken dar, die schließlich auf der Benutzeroberfläche des Werkzeugs visualisiert werden. Dieser Abschnitt beschreibt detailliert die Metrik-Berechnung der zentralen Sankey-Diagramm-Darstellung des Werkzeugs, das die Änderungen der Ticketeigenschaften visualisiert (im Folgenden Sankey-Metrik genannt). Die Java-Klasse *SankeyCalculatorBean*, die den hier beschriebenen Algorithmus implementiert, ist in Anhang A.3 wiedergegeben.

Die Sankey-Metrik erhält als Eingabe die im GUI ausgewählte Datenquelle, die zu analysierende Ticketeigenschaft sowie eine Liste von Filtern, die Einschränkungen

für die zu analysierenden Ticket-Daten definieren. Sie berechnet daraus eine JSON-Darstellung des Sankey-Graphen, die auf der grafischen Benutzeroberfläche durch die D3-Sankey-Javascript-Bibliothek (siehe Abschnitt 2.4) visualisiert wird. Der Berechnungs-Algorithmus der Metrik ist dabei wie folgt aufgebaut:

Zunächst wird aus dem Datenmodell eine Liste von Tickets bestimmt, auf die die gesetzten Filter zutreffen.

Die Filterung wird über das Strategie-Entwurfs-Muster[GHJV95] realisiert (Abbildung 5.7): Der *FilterContext* erhält dazu eine Liste aller Tickets der Datenquelle sowie eine Liste der Filter. Jeder Filter besitzt einen *Matcher*, der eine Methode *match()* bereitstellt, mit deren Hilfe der *FilterContext* entscheidet, ob ein Ticket herausgefiltert wird oder nicht.

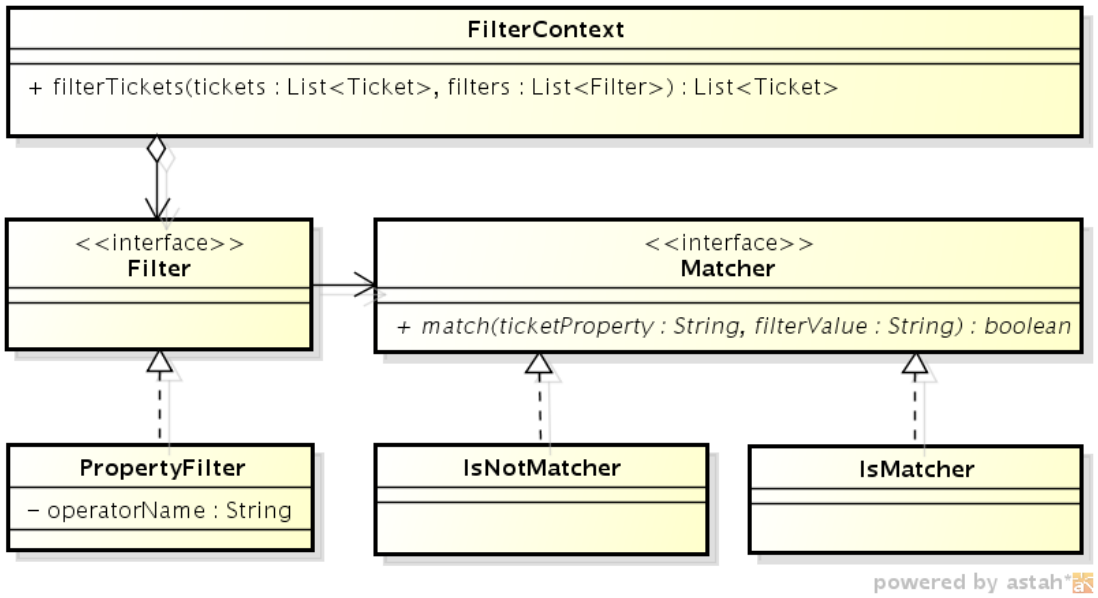

Abbildung 5.7: Klassendiagramm der Filterung

Aus dem Datenmodell wird nun eine Liste derjenigen Ticket-Änderungen ermittelt, die sich auf die zu analysierende Ticketeigenschaft beziehen und ein Ticket der gefilterten Ticket-Liste betreffen.

Aus dieser Liste wird nun eine Java-HashMap (*ticketChangeMap*) aufgebaut, die jedes Ticket auf seine Historie bezüglich der zu analysierenden Ticketeigenschaft abbildet. Dazu wird die Ticket-Id als Schlüssel und eine Liste von Ticketeigenschafts-Werten (beginnend beim ältesten und endend mit dem aktuellen Wert der Ticketeigenschaft) als Wert der HashMap verwendet. Tabelle 5.1 zeigt dies anhand eines (mit nur zwei Tickets sehr kleinen) Beispiels.

Nun wird unter Verwendung der Java-Graph-Modellierungs-Bibliothek jgrapht ein Modell des Sankey-Diagramm-Graphen erzeugt. Zunächst werden die Knoten erstellt: Aus jedem Eintrag aus der *ticketChangeMap* wird für jedes Präfix[6] der Ticket-Historie ein Knoten erstellt und nach dem Präfix benannt, falls ein Knoten mit diesem Namen

[6]hiermit sind unechte Präfixe gemeint, d.h. alle echten und auch die gesamte Ticket-Historie

Ticket-Id (Long)	Ticket-Historie (LinkedList<String>)
15	"new","assigned","accepted","closed"
16	"new","assigned","closed"

Tabelle 5.1: HashMap *ticketChangeMap*

noch nicht existiert. Zu jedem Knoten wird außerdem festgehalten, welche Tickets durch ihn dargestellt werden.

Aus der in Tabelle 5.1 dargestellten ticketChangeMap entstehen so 5 Knoten: “new(15,16)”, “new,assigned(15,16)”, “new,assigned,accepted(15)”, “new,assigned,accepted,closed(15)” und “new,assigned,closed(16)”.

Nach der Knoten-Erstellung werden diese mit gewichteten Kanten verbunden. Dazu iteriert der Algorithmus erneut über die Ticket-Historien aus der *ticketChangeMap*. Jedes Präfix mit zwei oder mehr Elementen stellt eine gerichtete und gewichtete Kante zwischen einem Quellknoten, der nach dem Präfix ohne dessen letztes Element benannt ist, und einem Zielknoten, der wie das Präfix heißt, dar. Existiert noch keine solche Kante im Graph-Modell, wird eine neue Kante mit Gewicht *eins* eingefügt, andernfalls wird das Gewicht der existierenden Kante um *eins* erhöht. Wie zu den Knoten werden zu jeder Kante die IDs der durch sie repräsentierten Tickets notiert. Tabelle 5.2 zeigt die vier Kanten, die aus obiger Ticket-Historie entstehen.

Quellknoten	Zielknoten	Kantengewicht	Ticket-Ids
"new"	"new","assigned"	2	15,16
"new","assigned"	"new","assigned", "accepted"	1	15
"new","assigned", "accepted"	"new","assigned", "accepted","closed"	1	15
"new","assigned"	"new","assigned", "closed"	1	16

Tabelle 5.2: Kanten des Sankey-Graph-Modells

Der letzte Schritt des Algorithmus erzeugt die JSON-Darstellung des Graph-Modells. Wie in Abschnitt 2.4 beschrieben, besteht die JSON-Darstellung aus einer Menge mit zwei benannten Elementen (den Listen *nodes* und *links*). Die Knoten werden in der JSON-Darstellung nach dem letzten Element des Knotennamens in der Graph-Modell-Darstellung benannt. Aus z.B. dem Knotennamen “new, assigned, accepted, closed” im Graph-Modell entsteht der Knotenname “closed” in der JSON-Darstellung. Zusätzlich besitzen sie eine Liste ihrer Ticket-Ids. Zu den Kanten werden die Indizes des Quell- und des Zielknotens, das Kantengewicht und ebenfalls die repräsentierten Ticket-Ids notiert. Quelltext 5.5 zeigt die JSON-Darstellung, die aus dem obigen Graph-Modell erzeugt wird. Abbildung 5.8 schließlich zeigt die daraus entstehende Visualisierung durch die D3-Sankey-Diagramm-Bibliothek.

```
{
  "nodes":[
     {"name":"new","tickets":[15,16]},
     {"name":"assigned","tickets":[15,16]},
     {"name":"accepted","tickets":[15]},
     {"name":"closed","tickets":[15]},
     {"name":"closed","tickets":[16]}
  ],
  "links":[
     {"source":0,"target":1,"value":2.0,"tickets":[15,16]},
     {"source":1,"target":2,"value":1.0,"tickets":[15]},
     {"source":2,"target":3,"value":1.0,"tickets":[15]},
     {"source":1,"target":4,"value":1.0,"tickets":[16]}
  ]
}
```

Quelltext 5.5: JSON-Darstellung des Sankey-Diagramms

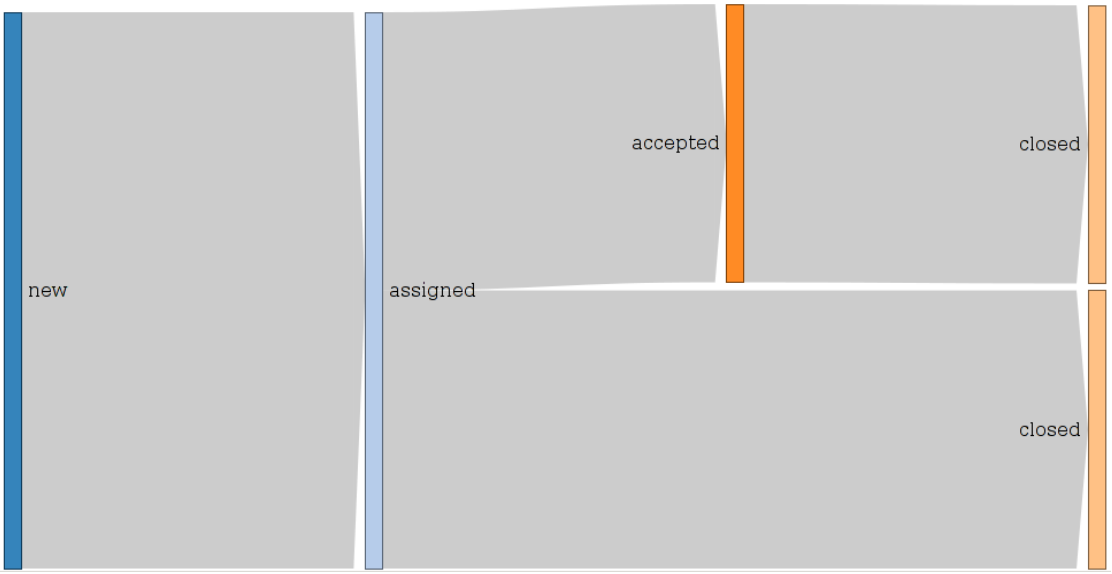

Abbildung 5.8: Visualisierung der Ticket-Änderungen im Sankey-Diagramm

5.5 Daten-Import und -Aktualisierung

Dieser Abschnitt beschreibt die für den Daten-Import aus und die Daten-Aktualisierung durch Change-Request-Systeme verantwortlichen Komponenten des Werkzeugs. Der Entwurf dieser Komponenten wurde insbesondere durch zwei Ziele beeinflusst:

1. Der Entwurf soll eine mögliche zukünftige Integration der Komponenten in MeDIC-Dashboard unterstützen.

2. Der Import soll aus verschiedenen Change-Request-Systemen möglich sein (vgl. Anforderung Q4).

Um eine Integration der Daten-Import- und Daten-Aktualisierungs-Komponenten in MeDIC-Dashboard zu ermöglichen, nutzt dieses Werkzeug das "Architekturmodell zur Integration heterogener Systeme in MeDIC"[Ste13], das A. Steffens im Rahmen seiner Diplomarbeit entworfen hat, als Basis für den Entwurf dieser Komponenten. Das Architekturmodell beschreibt die nötige Infrastruktur und die Rollen der verschiedenen Komponenten, die an einer Anbindung externer Systeme an MeDIC beteiligt sind. Insbesondere adressiert das Modell die Heterogenität der externen Systeme und der von ihnen verwendeten Datenformate, wie sie auch bei der Anbindung von unterschiedlichen Change-Request-Systemen und der Integration ihrer Daten in *River* vorliegt.

Im Zentrum des Architekturmodells steht der Enterprise Service Bus (ESB). Er erlaubt den Komponenten über standardisierte Nachrichten untereinander zu kommunizieren. Die von A. Steffens angeführten Besonderheiten des ESB ([Ste13], Kapitel 7.4.2) im Kontext von MeDIC gelten auch hier: Über den Nachrichtenbus werden ausschließlich Daten übertragen (Ticket-Daten und Ticket-Historien), nicht aber z.B. Administrations- oder Kontroll-Kommandos. Weiter ist die Richtung des Nachrichten-Flusses unidirektional und verläuft von den Modulen *initiale Ticket-Daten-Erfassung* und *XMLRPC-Webservice zur Ticket-Daten-Aktualisierung*, die Nachrichten erzeugen, zum Modul *Ticket-Daten-Verarbeitung*, das als Empfänger fungiert (vgl. Abbildung 5.2). Es findet außerdem kein Routing statt: Gesendete Nachrichten können von allen Teilnehmern des ESB empfangen und ausgewertet werden.

Ein elementarer Vorteil der Bus-Architektur ist die Entkopplung der Kommunikationsteilnehmer. Es existieren keine direkten Abhängigkeiten zwischen den Sendern und den Empfängern. Lediglich die Struktur und Semantik der Nachrichten ist allen Teilnehmern bekannt. So können einerseits leicht weitere Change-Request-Systeme an *River* angebunden werden indem weitere *Adapter* (vgl. [Ste13], Kap 7.5.3) entwickelt werden. Andererseits können ebenso leicht weitere Empfänger an den Bus angeschlossen werden, sollten etwa die extrahierten Ticket-Daten der Change-Request-Systeme auch für andere Metriken im MeDIC-System von Interesse sein.

River nutzt den Java Message Service (JMS) zur technischen Realisierung der Bus-Architektur.

Nachrichtentypen und Ticket-Daten-Verarbeitung

Die *Ticket-Daten-Verarbeitung* ist ein Nachrichten konsumierender *Adapter*. Die über die Nachrichten empfangenen Ticket-Daten werden normalisiert und in das Daten-Modell des Werkzeugs eingepflegt.

River kennt zwei Nachrichtentypen. Ihr Struktur und ihre durch die *Ticket-Daten-Verarbeitung* implementierte Semantik ist wie folgt definiert:

TicketMessage

Diese Nachricht enthält die nötigen Informationen zur eindeutigen Identifizierung eines Tickets (Name der Datenquelle und Ticket-Id), sowie die Eigenschaften des Tickets. Sie dient der Aktualisierung der ggf. bereits im Daten-Modell des Werkzeugs vorhandenen Daten zu diesem Ticket. Trifft eine *TicketMessage* in der *Ticket-Daten-Verarbeitung* ein, werden die Eigenschaften des Tickets im Daten-Modell durch die in der Nachricht übertragenen Eigenschaften und Eigenschaftswerte ersetzt. D.h. der in der Nachricht

beschriebene Ticketzustand entspricht dem neuen aktuellen Ticketzustand im Daten-Modell. Zusätzlich wird für jeden Unterschied (entfernte, geänderte bzw. hinzugefügte Eigenschaft) zum vorherigen Ticketzustand je eine Ticket-Änderung im Datenmodell erzeugt. Existiert das durch die Nachricht gekennzeichnete Ticket noch nicht im Daten-Modell, wird es erstellt und repräsentiert ein neues Ticket ohne Änderungs-Historie. Der Nachrichtentyp *TicketMessage* wird vom *Datei-Import-Adapter* und dem *XMLRPC-Webservice zur Ticket-Daten-Aktualisierung* verwendet.

TicketJournalMessage

Diese Nachricht dient dem (Re-)Import des aktuellen Zustands eines Tickets und seiner Historie. Sie besteht aus einer Liste, deren Einträge Kopien des Tickets in früheren und dem aktuellen Zustand darstellen. Der erste Eintrag in der Liste entspricht dem Zustand des Tickets bei seiner Erstellung. Der zweite Eintrag zeigt das Ticket nach der ersten Änderung usw. Der letzte Eintrag repräsentiert den aktuellen Zustand des Tickets. Beim Erhalt dieser Nachricht werden sämtliche bereits vorhandene Informationen über ein Ticket (aktueller Zustand und Historie) gelöscht. Anschließend iteriert die *Ticket-Daten-Verarbeitung* über die Einträge der Liste und erzeugt für jede Differenz (entfernte, geänderte oder hinzugefügte Eigenschaft) zwischen zwei Einträgen je eine Ticket-Änderung im Daten-Modell. Zuletzt wird der letzte Eintrag der Liste als aktueller Zustand des Tickets in das Daten-Modell aufgenommen. Der *Trac-Adapter* und der *Redmine-Adapter* verwenden diesen Nachrichtentyp.

Technisch wurde die *Ticket-Daten-Verarbeitung* durch zwei Message-Driven-Beans realisiert, die für den Empfang jeweils eines der beiden Nachrichtentypen verantwortlich sind. Anhang A.5 zeigt die Implementierung des Empfangs einer *TicketJournalMessage*.

Initiale Ticket-Daten-Erfassung

Das Modul *initiale Ticket-Daten-Erfassung* beinhaltet zu jedem unterstützen Change-Request-System einen *Adapter*: In diesem *Adapter* ist der Zugriff auf die Ticket-Daten des Change-Request-Systems implementiert. Die extrahierten Ticket-Daten werden anschließend in Nachrichten über den ESB versendet. Es wurden drei *Adapter* entwickelt:

Trac-Adapter

Der Trac-Adapter extrahiert die Ticket-Daten und Ticket-Historien über die XMLRPC-Schnittstelle des Trac-Change-Request-Systems[7]. Dazu verwendet er die Bibliothek tracdrops[8], die eine Java-Implementierung dieser Schnittstelle zur Verfügung stellt. Er beherrscht sowohl den anonymen als auch den authentifizierten Zugriff auf das Trac-System. Jedes abgerufene Ticket und seine Historie wird zu einer Liste von Ticketzuständen aufbereitet und in einer *TicketJournalMessage* versandt.

[7] `http://trac-hacks.org/wiki/XmlRpcPlugin`, abgerufen am 09.06.2013
[8] `http://code.google.com/p/tracdrops/`, abgerufen am 09.06.2013

Redmine-Adapter

Der Redmine-Adapter arbeitet analog zum Trac-Adapter, und nutzt ebenfalls die entsprechende XMLRPC-Schnittstelle[9]. Er verwendet die Java-Implementierung redmine-java-api[10].

Datei-Import-Adapter

Der Datei-Import-Adapter dient dem Import eines aus einem Change-Request-System in eine Export-Datei exportierten Schnappschusses der Ticket-Zustände zum Export-Zeitpunkt. Die Export-Datei liegt dabei in einem (Textdatei)-Format vor, das folgende Eigenschaften besitzt:

- Ein Trennzeichen trennt Eigenschaftsnamen und Eigenschaftswerte. Es tritt innerhalb der Eigenschaftsnamen und Eigenschaftswerte nicht auf.
- Die erste Zeile (Titelzeile) der Datei enthält die Namen der Eigenschaften eines Tickets. Jeder Eigenschaftsname ist eindeutig, es gibt keine Duplikate.
- Jede weitere Zeile (Ticketzeilen) definiert den Zustand eines Tickets, indem in jeder Spalte der Wert der entsprechenden Ticketeigenschaft notiert ist. Jede Zeile besitzt genau so viele Spalten wie die Titelzeile. Leere Ticketeigenschaften resultieren in direkt aufeinanderfolgenden Trennzeichen.
- Es gibt eine Spalte, die die Ticket-Id repräsentiert. Der Eigenschaftsname ist frei wählbar, der in dieser Spalte angegebene Eigenschafts-Wert jedes Tickets ist numerisch.
- Es gibt eine Spalte, die das Ticket-Thema repräsentiert.

Der Datei-Import-Adapter erzeugt aus jeder Ticketzeile eine *TicketMessage*, die einen neuen Zustand dieses Tickets repräsentiert und das Datenmodell entsprechend aktualisiert. Auf diese Weise können sukzessiv mehrere Export-Dateien (z.B. monatliche Abzüge), die den Status der Tickets im Change-Request-System zu verschiedenen, aufeinanderfolgenden Zeitpunkten beschreiben, importiert werden. Im Daten-Modell des Werkzeugs entsteht auf diese Weise eine Ticket-Historie, die den Zustand der Tickets zu den Exportzeitpunkten charakterisiert. Multiple Ticket-Änderungen derselben Ticket-Eigenschaft zwischen zwei Exportzeitpunkten werden mit diesem Verfahren allerdings nicht erfasst. Sie verschmelzen zu einer einzelnen Ticket-Änderung. Der Datei-Import-Adapter eignet sich daher für Change-Request-Systeme, die keine Schnittstelle zur Abfrage der Ticket-Daten und Ticket-Historien anbieten oder bei denen diese Schnittstelle deaktiviert ist.

Ticket-Daten-Aktualisierung

Nach einem Ticket-Daten-Import aus einem Change-Request-System spiegelt das Daten-Modell des Werkzeugs den Zustand und die Historie der Tickets zum Import-Zeitpunkt

[9] `http://www.redmine.org/projects/redmine/wiki/Rest_api`, abgerufen am 09.06.2013
[10] `https://github.com/taskadapter/redmine-java-api`, abgerufen am 09.06.2013

wider. Ohne weitere Möglichkeiten zur Übertragung der Ticket-Daten aus Change-Request-Systeme in das Daten-Modell des Werkzeugs wird dieser Import- bzw. Re-Import-Vorgang vom Benutzer des Werkzeugs typischerweise in regelmäßigen zeitlichen Abständen oder vor einem Analyse-Vorgang initiiert. Der wesentliche Nachteil dieser Vorgehensweise besteht darin, dass ein voraussichtlich großer Anteil (da sich die existierende Historie der Tickets eines Change-Request-Systems nicht ändert) der übertragenen Ticket-Daten aus einer bereits zuvor genutzten Datenquelle bereits im Werkzeug vorhanden ist. Lediglich die Änderungen seit dem letzten Import-Vorgang stellen neue Informationen dar. Abhängig von der Menge der in einem Change-Request-System enthaltenen Daten, der Art des Imports (Abruf über eine XMLRPC-Schnittstelle oder Import aus einer zuvor exportierten Ticket-Daten-Datei) und der Geschwindigkeit der Netzwerkanbindung entsteht durch einen Import-Vorgang ggf. erheblicher Zeit- und Ressourcenaufwand.

Die *Ticket-Daten-Aktualisierung* adressiert dieses Problem, indem es eine Schnittstelle für Change-Request-Systeme anbietet, über die diese das Werkzeug aktiv über Ticket-Daten-Änderungen informieren können. Nach dem erstmaligen Import der Ticket-Daten aus einer Datenquelle werden nachfolgende Änderungen im Change-Request-System über diese Schnittstelle übertragen. Diese Übertragung wird durch das Change-Request-System initiiert. So wird die Übertragung redundanter Daten vermieden, und das Daten-Modell des Werkzeugs kontinuierlich mit den Change-Request-Systemen synchronisiert. Die *Ticket-Daten-Aktualisierung* ist somit ein *Gateway* in der Terminologie des Architekturmodells nach A. Steffens[Ste13].

Technisch wurde die *Ticket-Daten-Aktualisierung* durch die Implementierung eines XMLRPC-Dienstes realisiert. Über die Methode “ticket.update”, die der Dienst anbietet, kann ein Change-Request-System das Werkzeug über Ticket-Änderungen benachrichtigen. Dazu übergibt es der XMLRPC-Methode Informationen zur Identifizierung des betroffenen Tickets (Datenquelle und Ticket-Id) sowie die aktuellen Ticketeigenschaften des Tickets. Nach einer Validitäts-Prüfung der empfangenen Daten erzeugt die *Ticket-Daten-Aktualisierung* eine *TicketMessage* und sendet sie über den Nachrichtenbus des Werkzeugs. Diese wird wie oben beschrieben von der *Ticket-Daten-Verarbeitung* entgegengenommen.

Weiter wurde exemplarisch ein Plugin für das Change-Request-System Trac entwickelt, das die Nutzung der XMLRPC-Schnittstelle demonstriert. Anhang A.6 zeigt die Implementierungen des XMLRPC-Diensts zur *Ticket-Daten-Aktualisierung* und des Trac-Plugins.