Fakultät für Mathematik, Informatik und Naturwissenschaften

Forschungsgruppe Softwarekonstruktion

Diplomarbeit

Entwurf eines generischen Prozessleitstandes für Change Request Systeme

Development of a Generic Process Dashboard for Change Request Systems

Christian Charles

12. Juni 2013

Gutachter
Prof. Dr. rer. nat. Horst Lichter
Prof. Dr. rer. nat. Bernhard Rumpe

Betreuer
Dipl.-Inform. Matthias Vianden

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 12. Juni 2013 (Christian Charles)

Einige Worte des Dankes

An dieser Stelle möchte ich mich recht herzlich bei allen bedanken, die mir auf vielfältige Weise Gelegenheit gaben, diese Diplomarbeit zu erstellen:

Herr Prof. Dr. rer. nat. Horst Lichter überließ mir das Thema und unterstützte mich während der Entwicklung der Arbeit.

Herrn Prof. Dr. rer. nat. Bernhard Rumpe danke ich für die Übernahme des Zweitgutachtens.

Bei Herrn Dipl.-Inform. Matthias Vianden möchte ich mich für die Betreuung dieser Diplomarbeit bedanken. Seine stets konstruktiven und zielführenden Hinweise waren bei der Bearbeitung des Themas gleichermaßen hilfreich wie inspirierend und bestimmten wesentlich den Fortgang dieser Arbeit.

Zum Schluss danke ich meinen Eltern, die mich während des Studiums stets unterstützt haben.

Christian Charles

Kurzdarstellung

Deutsch

Die Analyse und Optimierung von Softwareentwicklungs-Prozessen ist eine komplexe und Ressourcen-intensive Herausforderung für Software-entwickelnde IT-Organisationen. Softwareentwicklungs-Prozesse werden durch die Verwendung von Standard-Software-Werkzeugen wie Change-Request-Systemen unterstützt. In dieser Diplomarbeit wird der Frage nachgegangen, inwiefern Daten, die durch die Nutzung solcher Werkzeuge entstehen, zur Analyse der Softwareentwicklungs-Prozesse genutzt werden können, die diese Werkzeuge einsetzen.

English

Analyzing and optimizing software development processes is a complex and resource demanding task faced by organizations which are concerned with software development. Software development processes are supported by the application of standard software tools such as change request systems. This diploma thesis considers the question, to what extent data, that is generated due to the usage of such tools, can be exploited for the purpose of analyzing the software development processes, which apply these tools.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Liste der Quelltexte

1 Einleitung

Inhaltsangabe

Seit Beginn der Softwarekrise Mitte der 1960er Jahre stellt sich immer wieder die Frage: Wie stellt man sicher, dass ein Softwareprojekt erfolgreich durchgeführt wird[Rum10]? Oder, etwas schwächer formuliert in Anerkennung der empirischen Erkenntnis, dass der Stein der Weisen noch nicht gefunden wurde und auch und gerade heute Softwareprojekte weiterhin scheitern [Cha05]: Wie erhöht man zumindest die Erfolgs-Wahrscheinlichkeit? Die Komplexität von Software steigt immer weiter und damit das Risiko des Scheiterns. Zahlreiche Forschungsprojekte haben das Ziel, dieses Risiko zu mindern. Softwareentwicklungs-Prozesse wie etwa das Spiral Modell, der Rational Unified Process, Extreme Programming[LL10] und Scrum[1] strukturieren den Vorgang der Softwareentwicklung. Sie definieren Rollen, die Mitarbeiter einnehmen, sowie Aktivitäten und Artefakte (Dokumente), die diese durchzuführen bzw. zu erstellen haben. Auf diese Weise wird das komplexe Software-Projekt in handhabbarere, kleinere Teile zerlegt. Prozessverbesserungs-Modelle wie CMMI oder ISO9000 bewerten Organisationen daraufhin, wie gut sie ihren definierten Prozessen folgen[2].

Sowohl Softwareentwicklungs-Prozesse als auch Prozessverbesserungs-Modelle bedienen sich Metriken, um Aussagen über Qualitätseigenschaften der zugrundeliegenden Entität zu treffen. Erstere nutzen Metriken wie etwa Lines of Code, zyklomatische Komplexität oder Testabdeckung, um Eigenschaften von Software zu bewerten. Das Prozessverbesserungs-Modell CMMI nutzt sogenannte SCAMPI-Untersuchungen, in denen festgestellt wird, in welchem Reifegrad sich die in der Organisation eingesetzten Prozesse befinden. Während die Metriken für Software bereits ausgiebig erforscht sind und mit Werkzeugunterstützung (z.B. Sonar[3]) versehen sind, ist insbesondere Letzteres für Metriken auf Softwareentwicklungs-Prozessen Mangelware.

1.1 Ziel der Arbeit

Im Rahmen dieser Diplomarbeit soll herausgefunden werden, ob und inwiefern die Datenerhebung und Darstellung von Metriken für Softwareentwicklungs-Prozesse automatisiert werden kann und inwiefern diese Metriken hilfreiche Erkenntnisse über die Umsetzung des Softwareentwicklungs-Prozesses liefern.

[1] `http://www.scrumalliance.org/`

[2] Übersetzt von `http://en.wikipedia.org/wiki/Software_development_process`: "Independent assessments grade organizations on how well they follow their defined processes"

[3] `http://www.sonarsource.org/`

"You can't even ask them to push a button" (Man kann sie nicht einmal darum bitten, einen Knopf zu drücken)[Joh01]: Mit diesem prägnanten Titel stellt Johnson die Abneigung von (nicht nur) Entwicklern heraus, zusätzliche Aktivitäten wie etwa Ist- und Soll-Aufwand-Erfassung durchzuführen, die nur "für die Qualitätssicherung" gut sind und keinen direkten Nutzen für den Entwickler zu haben scheinen. Er fand heraus (auch aus eigener Erfahrung), dass selbst nur einen Knopf zu drücken unzumutbar sei ("too much overhead"). Dieses Hindernis wird durch nicht-störende ("non-disruptive") Werkzeuge, d.h. Werkzeuge, die ohne aktive Interaktion mit dem Entwickler funktionieren, aus dem Weg geräumt.

Softwareentwicklungs-Prozesse, so die Idee dieser Arbeit, spiegeln sich in den Werkzeugen wider, die von den Entwicklern ohnehin genutzt werden, wie etwa Change-Request-Systeme. Sieht etwa ein agiler Prozess wie Scrum vor, dass User Stories (und daraus resultierende Aufgaben) innerhalb eines Sprints zu erledigen sind, so kann man diese Aufgaben und ihren Fortschritt ("ausstehend", "in Bearbeitung", "erledigt") im Change-Request-System wiederfinden.

Kann man die Daten, die in diesen Werkzeugen entstehen, hinsichtlich einer Bewertung der Umsetzung des Prozesses sinnvoll auswerten? Die Change-Request-Systeme selbst bieten dafür wenig Unterstützung. Sie richten sich vor allem an ihre direkten Nutzer, die Entwickler. Daher soll während der Diplomarbeit ein Werkzeug entwickelt werden, das diese Informationen aufbereitet. In Evaluationen des Werkzeugs mit Projektleitern soll anschließend ermittelt werden, welche Daten und welche Darstellungen als sinnvoll und hilfreich erachtet werden. Erkennt man Probleme im Softwareentwicklungs-Prozess leichter? Welche Informationen aus den Change-Request-Systemen sind interessant und welche eher nicht?

2 Grundlagen

Inhaltsangabe

In diesem Kapitel werden die konzeptionellen und technologischen Grundlagen erläutert, auf denen die Diplomarbeit aufbaut.

2.1 Qualität

Das Verständnis des Begriffs Qualität ist für die Prozessbewertung und -verbesserung von zentraler Bedeutung. Er wird daher hier zunächst definiert und erläutert.

Qualität im Kontext von Software und Softwareentwicklungs-Prozessen grenzt sich deutlich von der Verwendung des Wortes im Alltag ab. Eine im Supermarkt mit "beste Qualität" beworbene Apfelsorte sorgt bei den Kunden nicht etwa für verwirrte Gesichter. Vielmehr werden die meisten damit implizit Eigenschaften wie "knackig", "lecker" oder "vitaminreich" verbinden.

Was aber bedeutet Qualität im Zusammenhang mit Software und Prozessen? Man spricht hier nicht schwammig von "gut" oder "schlecht". Der IEEE Standard 610.12 definiert Qualität wie folgt:

> 1. The degree to which a system, component, or process meets specified requirements
> 2. The degree to which a system, component, or process meets customer or user needs or expectations
>
> (IEEE Standard 610.12[IEE90])

Es geht demnach um klar definierte, explizit aufgeschriebene Anforderungen bzw. implizite Erwartungen der Benutzer. Die expliziten Anforderungen werden hierbei in den Anforderungsdokumenten notiert. Ein Beispiel für eine (sehr allgemeine) implizite Erwartungshaltung könnte etwa sein, dass sich eine Software mehr als einmal starten lässt, und nicht nach dem ersten Mal weggeworfen werden muss. Eine solche Anforderung ist gemeinsamer impliziter Konsens zwischen den Stakeholdern. Der Grad, zu dem diese - explizite oder implizite - Anforderung erfüllt ist, ist die Qualität *bezüglich dieser Anforderung.*

Software-Qualitäten teilen sich nach der üblichen Klassifikation in Produkt- und Prozessqualitäten, je nachdem ob sie eine Eigenschaft des Softwareprodukts oder des Softwareentwicklungs-Prozesses beschreiben. Bereits 1976 hat Boehm eine Typologie in Software-Projekten wiederkehrender Produktqualitäten aufgestellt[BBL76]. Er hat dazu die verschiedenen Qualitäten in einer Baumstruktur organisiert, und so Qualitäts-Oberbegriffe geschaffen, die auf grundlegenden Qualitäten aufbauen.

Die Qualität einer Software aus Nutzersicht beschreibt der Zweig *Brauchbarkeit*. Er enthält u.a. die *Zuverlässigkeit*, die sich wiederum aus *Genauigkeit*, *Vollständigkeit* und *Robustheit* zusammensetzt. Beispiele für diese Qualitäten sind etwa: "Berücksichtigt der Taschenrechner genügend Nachkommastellen?" (Genauigkeit), "Enthält die Textverarbeitung eine Druckfunktion?" (Vollständigkeit) und "Wie geht die Software mit Fehlbedienung um?" (Robustheit). Im zweiten Haupt-Zweig in Boehms Typologie, der *Wartbarkeit*, wird die Qualität der Software aus Herstellersicht betrachtet. Hier seien als Beispiele die *Änderbarkeit* und die *Testbarkeit* genannt. In gut strukturiertem, dokumentiertem Quellcode lassen sich leichter neue Anforderungen umsetzen oder Fehler beheben als in sogenanntem "Spaghetticode".

Neben diesen Produktqualitäten stehen die Prozessqualitäten[Lic11]. Sie beschreiben nicht direkt Eigenschaften der Software, sondern des Entwicklungs-Prozesses, mit dem die Software hergestellt wird. Zu ihnen gehören *Termin- und Aufwandseinhaltung*. Je genauer Aufwand und Termine korrekt eingeschätzt werden können, desto besser können etwa Preise festgelegt und das Risiko von Konventionalstrafen wegen Terminverzugs verringert werden. Aber auch *Bausteingewinn* (Entstehung bzw. Verbesserung von wiederverwendbaren Software-Komponenten) und *Know-How-Gewinn* gehören zu den Prozessqualitäten. Eine für diese Arbeit wichtige Prozessqualität ist die *Prozesstransparenz*. Sie beschreibt, inwieweit der Software-Entwicklungsprozess definiert ist und im Software-Projekt umgesetzt wird.

Prozess- und Produktqualitäten sind keine voneinander unabhängigen Merkmale. Die Prozessqualitäten beeinflussen die Produktqualitäten, im Positiven wie im Negativen. Ein Beispiel: Sieht der Prozess testgetriebene Entwicklung vor, dann werden Tests "konsequent vor den zu testenden Komponenten"[1] entwickelt. Die Idee dabei ist, die Produktqualität *Testbarkeit* der Software zu erhöhen, indem sichergestellt wird, dass Tests systematisch erstellt werden, und nicht etwa am Ende eines Projekts aufgrund von Termin- oder Kostendrucks zu kurz kommen. Gleichzeitig sinkt aber eine andere Produktqualität, die *Änderbarkeit*. Bei Änderungen an der Software, insbesondere bei grundlegenden Änderungen in Architektur und Design, müssen ggf. auch Änderungen an den Tests durchgeführt werden. Sind viele solcher Änderungen im Verlauf des Projekts zu erwarten, entsteht gegenüber einem Prozessmodell, das die Testerstellung erst am Ende vorsieht, deutlich mehr Aufwand. Im Allgemeinen, aber ohne Garantie, gilt: Eine hohe Prozessqualität fördert die Entstehung hochwertiger Produkte und umgekehrt[Lic11].

[1] `https://de.wikipedia.org/wiki/Testgetriebene_Entwicklung` abgerufen am 15.05.2013

2.2 Metriken

Während in der Vergangenheit der Erfolg eines Software-Projekts maßgeblich von dem Genie einzelner Akteure abhängig war, lässt sich die Informatik, und insbesondere ihr Teilbereich Softwaretechnik[Rum10], heutzutage von bewährten Methoden aus anderen Wissenschaften, vor allem den Ingenieurswissenschaften inspirieren.

> "Software Engineering zielt auf die ingenieurmäßige Entwicklung, Wartung, Anpassung und Weiterentwicklung großer Softwaresysteme unter Verwendung bewährter systematischer Vorgehensweisen, Prinzipien, Methoden und Werkzeuge."
>
> Manifest der Softwaretechnik, 2006

Eines dieser bewährten Prinzipien ist das Messen. Nicht ein diffuses Bauchgefühl oder der Erfahrungsschatz einzelner Mitarbeiter sollen über Erfolg oder Misserfolg eines Software-Projekts bestimmen. Vielmehr sollen Messungen Auskunft über den Zustand der Software bzw. des Softwareentwicklungs-Prozesses geben und so die Grundlage für Entscheidungen und Maßnahmen liefern, um die Risiken in der Softwareentwicklung zu mindern und Qualitäten zu verbessern. Spätere Messungen indizieren dann den wiederum Erfolg dieser Entscheidungen und Maßnahmen.

"[You] cannot control what you cannot measure"[DeM86]: DeMarco, ein früher Verfechter von Software-Metriken, hat seine Aussagen über die Unabdingbarkeit von Messungen für den Erfolg von Software-Projekten mittlerweile zwar relativiert[DeM09]. Er führt an, dass bei einem sehr lukrativem Projekt, das 1 Million Dollar kostet, aber 50 Millionen erwirtschaftet, Messungen und Controlling kaum eine Rolle spielen. Selbst wenn es aufgrund schlechtem Managements zehnmal so teuer wird, kann es als Erfolg angesehen werden wenn das Ziel lautete, Gewinn gemacht zu haben. Bei einem Projekt, das bei 1 Million Dollar Kosten nur 1,1 Millionen abwirft, ist es hingegen sehr wichtig, dass Kosten- und Zeitpläne eingehalten werden.

Auch nur indirekt oder schwer messbare Dinge entziehen sich nicht vollkommen der Kontrolle. Dazu werden plausible Kausalitäts-Annahmen getroffen. Dieses Konzept ist aus der Mess- und Regelungstechnik als "Steuerung (engl. open loop control)" bekannt. Eine Eingangs- bzw. Führungsgröße bestimmt die Stellgröße, wodurch wiederum die zu steuernde Größe beeinflusst wird. Die Führungsgröße entspricht hier dem gesetzten Qualitätsziel, die Stellgröße der Qualitätsmaßnahme, und die zu steuernde Größe ist die Qualität. In der Steuerung wird ein klares Modell des Einflusses der Stellgröße auf die zu steuernde Größe benötigt. Beispiel: Eine Heizung soll eine Flüssigkeit von einer Ausgangstemperatur auf eine Zieltemperatur aufheizen. Ist die Menge der Flüssigkeit, deren Wärmekapazität und die Heizleistung der Heizung bekannt, kann aus dem physikalischen Modell die nötige Heizdauer ermittelt werden, ohne dass gemessen werden muss, ob die Zieltemperatur erreicht wurde.

In Softwareentwicklungs-Prozessen ist das den Qualitätsmaßnahmen zugrunde liegende Modell in der Regel nicht so klar verstanden wie ein physikalisches Modell. Eine gute Kaffee-Versorgung und ein Tischkicker im Pausenraum, so könnte z.B. eine Annahme über den Einfluss einer Qualitätsmaßnahme lauten, erhöhen die Mitarbeiterzufriedenheit. Ist diese wiederum hoch, so steigen auch diverse andere Qualitäten. Zufriedene Mitarbeiter verbleiben eher im Unternehmen, es verringert sich also das Risiko von

Know-How-Verlust. Vielleicht sinkt aber auch die Produktivität, weil der Tischkicker so laut ist und die Mitarbeiter, die gerade keine Pause einlegen in der Konzentration stört.

Daher ist es wichtig, die Auswirkungen von Qualitätsmaßnahmen zu messen, und so den Grad ihrer Wirksamkeit festzustellen. Messungen zeigen quantitativ auf, ob und zu welchem Grad das gesetzte Ziel erreicht bzw. nicht erreicht wurde.

Wie aber können nicht greifbare, virtuelle Dinge wie Software oder Prozesse vermessen werden? Dazu dienen Metriken, die das IEEE wie folgt definiert[IEE90]:

metric

A quantitative measure of the degree to which a system, component, or process possesses a given attribute.

quality metric

1. A quantitative measure of the degree to which an item possesses a given quality attribute.
2. A function whose inputs are software data and whose output is a single numerical value, that can be interpreted as the degree to which the software possesses a given quality attribute.

Eine Metrik bildet also eine Eigenschaft (Metrik-Attribut) der Software oder des Software-Prozesses (Gegenstand der Messung) auf einen Wert ab. Dieser Wert bedarf dann einer Interpretation und/oder Weiterverarbeitung, da es nicht offensichtlich ist, was sich aus dem gemessenen Wert schlussfolgern lässt. Ludewig und Lichter[LL10] klassifizieren Metriken nach diversen Kriterien. Für diese Diplomarbeit ist sowohl die Differenzierung zwischen Basismetriken und abgeleiteten Pseudo-Metriken als auch zwischen objektiver und subjektiver Metrik wichtig.

Des Weiteren definiert der ISO/IEC-Standard 15939[ISO07] das Maß-Informations-Modell, das den Zusammenhang zwischen Metrik-Attributen, Basis- und Pseudometriken sowie dem daraus resultierenden Informationsgewinn herstellt (siehe Abbildung 2.1). Dieser Zusammenhang und die eingeführten Metrik-Begriffe werden im Folgenden genauer erläutert.

Objektive Metriken werden nach eine Mess-Vorschrift automatisch oder manuell nach einem exakten Verfahren gemessen. Im Fall einer subjektiven Metrik wird der Wert geschätzt, und damit eine Beurteilung der von der Metrik beschriebenen Eigenschaft der Software bzw. des Softwareentwicklungs-Prozesses gegeben.

Basismetriken werden direkt gemessen oder geschätzt, liefern allerdings in der Regel keine direkt interpretierbaren Ergebnisse. Ein Beispiel für eine Basismetrik ist LOC (Lines of Code). Mit dieser Metrik wird die Anzahl der Quelltextzeilen einer Software erfasst. Eine direkte Interpretation ist allerdings wenig sinnvoll. Was bedeutet z.B. eine hohe Anzahl von Quelltextzeilen? Weist sie auf eine hohe Komplexität der Software hin? Oder ist das Projekt nahezu fertiggestellt, weil schon so viel Quelltext produziert wurde?

Pseudometriken zeichnen sich dadurch aus, dass sie nicht direkt gemessen oder geschätzt werden werden, sondern mittels eines Algorithmus aus anderen Basis- bzw. Pseudometriken errechnet werden. Dies geschieht in der Regel, weil das Metrik-Attribut nicht direkt oder nur aufwändig messbar ist.

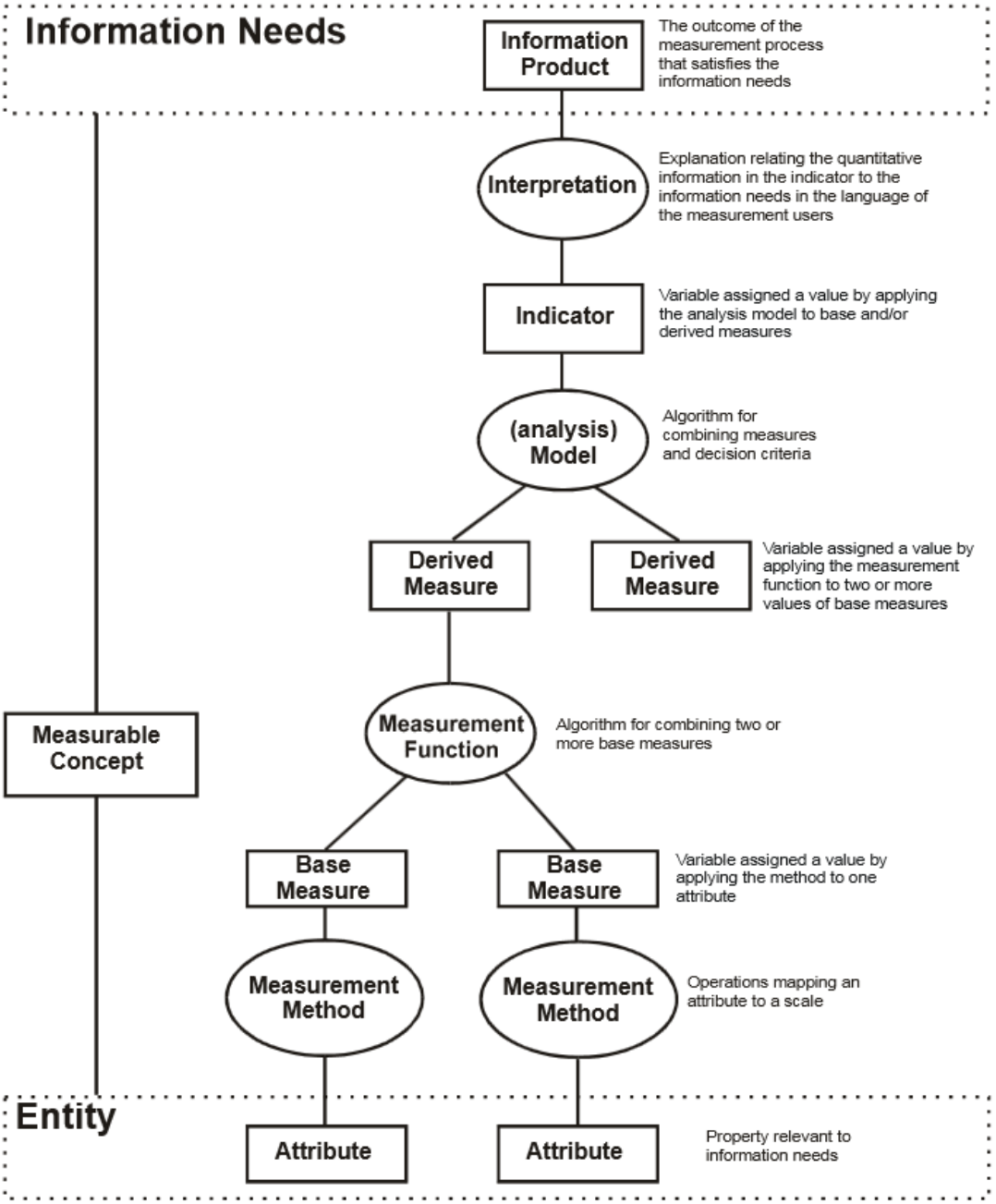

Abbildung 2.1: Maß-Informations-Modell nach ISO/IEC 15939

Metrik-Beispiel *Portabilität*

Als Beispiel für eine Pseudometrik soll die *Portabilität* dienen. Je geringer der Änderungsaufwand ist, um eine Software auf einer anderen Zielplattform auszuführen, umso höher ist die Portabilität. Ein einfacher Ansatz für eine solche Metrik kategorisiert die Quelltextzeilen in portable (bei einem C-Programm z.B. Variablendeklarationen, Kontrollstrukturen, Standard ANSI-C Befehle) und nicht portable Anweisungen (z.B. Aufrufe von Funktionen der Win32-API). Die Berechnungsvorschrift

$$\text{Portabilität} = \frac{\text{Anzahl der portablen Quelltextzeilen}}{\text{LOC}}$$

kalkuliert dann einen Wert zwischen Null und Eins, der das prozentuale Verhältnis portabler Quelltextzeilen zum Gesamtumfang der Software beschreibt. Die Interpretation lautet: Der Wert Eins bedeutet, dass die Software ohne Änderungen auf der neuen Zielplattform ausgeführt werden kann. Darüber hinaus gilt: Je näher der Wert an Eins ist, desto geringer ist der Änderungsaufwand zur Portierung der Software auf die neue Zielplattform.

Sehr wichtig bei der Definition von Metriken ist das Einhalten der Repräsentationsbedingung:

> Gilt eine Relation R im empirischen Relationssystem für das Attribut A, so muss auch die entsprechende Relation R' im numerischen Relationssystem gelten!

Im Fall der obigen Metrik Portabilität bedeutet dies: Eine leicht zu portierende Software sollte einen Wert nahe bei Eins erzielen und eine schwer zu portierende Software sollte deutlich unter Eins bewertet werden. Ist dies nicht der Fall, ist die Metrik "ungeeignet". Folgende drei Beispiele deuten darauf hin, dass die Metrik Portabilität, wie sie oben definiert wurde, tatsächlich nicht allgemein geeignet ist.

Portabel/Unportabel-Kriterium

Das Kriterium, nach dem Quelltextzeilen in die Kategorien portabel bzw. nicht portabel eingeordnet werden, ist stark von der bisherigen und der neuen Zielplattform abhängig. Die Metrik könnte etwa einer JavaEE-Anwendung, die bisher unter dem Applicationserver Glassfish[2] lief, und künftig auch unter dem Applicationserver WebSphere[3] ausgeführt werden soll, eine hohe Portabilität bescheinigen, weil nur wenige Glassfish-spezifische Quelltextzeilen verwendet wurden. Soll dieselbe Anwendung aber zu einer nativen Windowsanwendung werden, sind voraussichtlich erheblich mehr Änderungen nötig.

Verhältnismetrik

Die Metrik beschreibt den nötigen Änderungsaufwand relativ, und darf nicht mit einer absoluten Angabe verwechselt werden. Erhält Software A mit 10.000 Zeilen Quelltext

[2] `https://glassfish.java.net/` abgerufen am 20.05.2013
[3] `http://www-01.ibm.com/software/de/websphere/` abgerufen am 20.05.2013

den Portabilitätswert 0.99, so sind 100 Zeilen zu ändern. Software B mit dem selben Portabilitätswert, aber 10.000.000 Zeilen Quelltext, ist mit 100.000 Änderungen erheblich aufwändiger zu portieren.

Keine Gewichtung nach Schwierigkeit

Die Metrik setzt voraus, dass die Anpassung aller nicht portablen Zeilen gleich schwierig bzw. überhaupt möglich sei. Das ist i.A. nicht der Fall. Angenommen, eine Software, die die Temperatur einer Grafikkarte von Hersteller A überwacht, enthalte eine einzige nicht-portable Zeile: Diese liest über eine Funktion in der API des Grafikkartentreibers die Temperatur aus. Alle anderen Zeilen der Software, die etwa die regelmäßige Abfrage steuern, die Information visualisieren, oder den Benutzer bei Überschreiten einer Grenztemperatur warnen, seien portabel. Nun soll die Software auch mit einer Grafikkarte von Hersteller B funktionieren. Dieser stellt aber eine entsprechende Funktion zum Auslesen der Temperatur in der API seines Treibers überhaupt nicht zur Verfügung. Die Software ist somit überhaupt nicht portierbar, obwohl die Metrik einen hohen Portabilitätswert berechnet.

Diese Beispiele zeigen wie schwierig es ist, eine aussagekräftige Metrik zu definieren. Auf den ersten Blick erscheint die Berechnungsvorschrift zur Portabilität durchaus plausibel, zeigt dann aber Schwächen. Der Grund dafür liegt darin, dass die Definition von Metriken eine Modellbildung ist[Lic11]. Ein Modell veranschaulicht die modellierte Realität. Diese wird dabei auf die relevanten Aspekte vereinfacht. Fällt diese Vereinfachung zu stark aus, repräsentiert die Metrik nicht mehr die beabsichtigte Eigenschaft der Software bzw. des Softwareentwicklungs-Prozesses.

Metrik-Beispiele aus der Earned-Value-Analyse

Es gibt natürlich neben problematischen Metriken auch solche, die sich bewährt haben. Als Beispiel seien hier die Metriken der Earned-Value-Analyse genannt, die heutzutage regelmäßig und selbstverständlich im Projekt-Controlling eingesetzt werden. Die Basismetrik *Geplante Kosten (PV)* greift auf Daten des Projektplans zu Arbeitspaketen zurück, die Basismetrik *Tatsächliche Kosten (AC)* nutzt Daten aus der Arbeitszeit- und Ressourcen-Erfassung. Die Basismetrik *Erarbeiteter Wert (EV)* spiegelt eine Schätzung der Mitarbeiter wieder, zu welchem Grad ein Arbeitspaket fertiggestellt ist. Darauf bauen die Pseudometriken Kostenabweichung (CV) = EV − AC und Planabweichung (SV) = EV − PV auf. Aus diesen lässt sich quantitativ ableiten, inwieweit das Projekt dem Plan bezüglich der Kosten bzw. der Zeit hinterherhinkt oder voraus ist. Diese Metriken haben sich als geeignet erwiesen, um den Projektfortschritt zu beobachten. Ihre Aussagekraft hängt nur noch von der Datenqualität der zugrundeliegenden Datenquellen ab, also z.B. von der Vollständigkeit des Projektplans und der Zeiterfassung, sowie von der Güte der Schätzung des Arbeitspaket-Fertigstellungsgrads durch die Mitarbeiter. Da diese Metriken im weiteren Verlauf der Diplomarbeit keine Rolle spielen, sei der interessierte Leser für genauere Erläuterungen auf Kapitel *8.5 Projektkontrolle und -steuerung* im Buch *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*[LL10] verwiesen.

Zielgerichtetes Messen

Welche Metriken sollten in einer Organisation oder in einem Projekt erfasst und eingesetzt werden? Zur Beantwortung dieser Frage ist es hilfreich, sich noch einmal die Gründe für den Einsatz von Metriken vor Augen zu halten. Metriken sollen u.a. Führungskräfte dabei unterstützen, strategische wie auch akute Entscheidungen fundiert zu treffen. Offensichtlich gilt: Je mehr und genauere Informationen als Grundlage für eine Entscheidung vorliegen, desto besser. Demnach wäre auf die Frage "Welche Metriken?" die naheliegende Antwort: "Möglichst alle!". Dies gilt natürlich nur mit Einschränkungen, denn der Einsatz einer Metrik bedeutet Aufwand. Die zugrundeliegenden Daten müssen erfasst werden, die Metriken berechnet und schließlich interpretiert werden. Im Rahmen der oben genannten Metriken der Earned-Value-Analyse etwa müssen Mitarbeiter regelmäßig den Fortschritt von Arbeitspaketen einschätzen, eine Aktivität, auf die ohne diese Metrik verzichtet werden könnte. Der Einsatz einer Metrik ist daher ein Kostenfaktor, dem ein Nutzen gegenüberstehen sollte.

Der Goal-Question-Metric-Ansatz[BCR94] ist eine Methode, mit deren Hilfe eine Organisation bzw. ein Projekt zielgerichtet und systematisch ermitteln kann, welche Metriken sie/es sinnvollerweise einsetzen sollte: Der Ausgangspunkt und erster Schritt des Ansatzes ist die Festlegung der Ziele der Organisation bzw. des Projekts. Sie stammen aus dem Unternehmensleitbild bzw. werden zu Beginn eines Projekts festgelegt. Im zweiten Schritt werden zu diesen Zielen Fragen formuliert, deren Beantwortung Auskunft darüber gibt, inwieweit die gesetzten Ziele erreicht sind. Diese Fragen sind in der Regel nicht direkt von einer Metrik beantwortbar. Daher werden sie weiter in konkretere Fragen zerlegt, deren Antworten zusammen die übergeordnete Frage beantworten. Diese Vorgehensweise wird fortgesetzt, bis die Fragen Metriken zugeordnet werden können.

Ein Beispiel:

- Ziel:
  - G*1* Erhöhung der Kundenzufriedenheit
- Fragen:
  - Q*1* Wie zufrieden sind die Kunden mit dem Support?
  - Q*2* ...
- Unterfragen:
  - SQ*1* Wie schnell werden Kundenanfragen bearbeitet?
  - SQ*2* Wie hoch ist die Erfolgsquote bei der Lösung des Problems des Kunden?
  - SQ*3* ...
- Metriken:
  - M*1* Zeit vom Eingang der Kundenanfrage bis zu deren Lösung.
  - M*2* Anteil der Tickets, die nach einer Lösung wieder eröffnet werden.
  - M*3* Anteil der neuen Tickets, die ein Kunde zu einem bereits durch ein altes Ticket gelöst geglaubten Problems eröffnet.

M*4* Auswertung eines Feedback-Formulars, das Kunden nach einem Support-Vorgang ausfüllen können.

M*5* ...

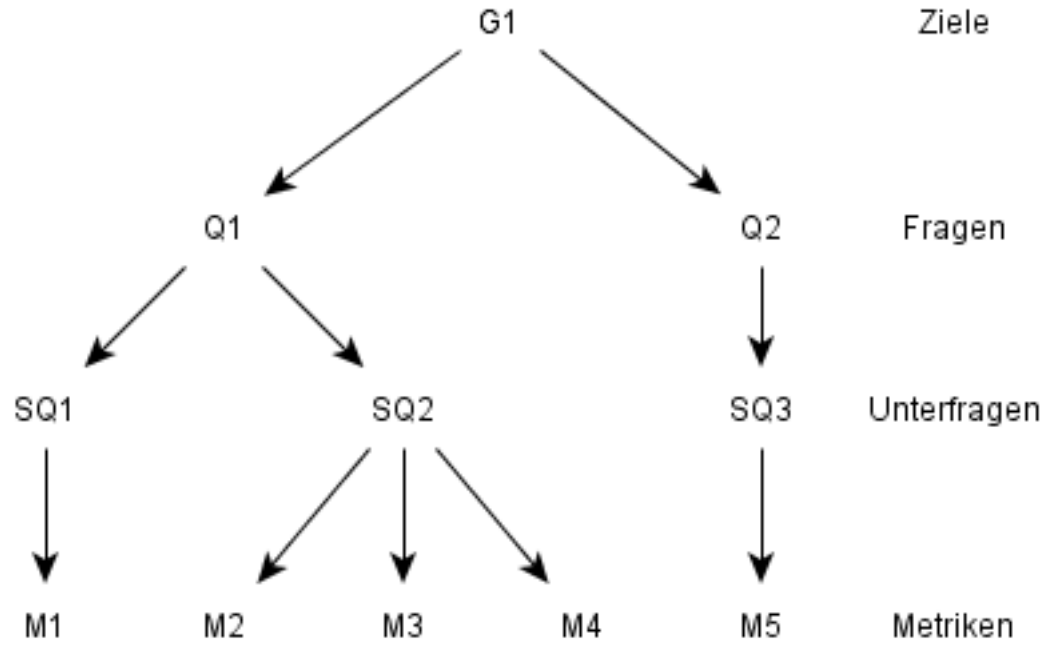

Abbildung 2.2: Goal-Question-Metric-Ansatz

Durch diesen Zerlegungsprozess entsteht von oben nach unten ein Baum, wie in Abbildung 2.2 dargestellt, der die Ziele der Organisation bzw. des Projekts mit Metriken verbindet. Der Vorteil dieser Methode liegt darin, dass die Organisation genau weiß, zu welchem Zweck sie eine konkrete Metrik erhebt, und so die Nutzung irrelevanter Metriken vermeidet.

In dieser Diplomarbeit wird ein anderer, gewissermaßen zu GQM umgekehrter, Bottom-Up-Ansatz verfolgt und daraufhin untersucht, inwieweit sich damit das Ziel, Softwareentwicklungs-Prozesse zu analysieren und zu optimieren, erreichen lässt. Im Rahmen dieses Bottom-Up-Ansatzes sollen Daten über den Softwareentwicklungs-Prozess, die bereits durch die Umsetzung des Prozesses anfallen, Metriken induzieren, die diese Daten auswerten. Diese Metriken wiederum beantworten entsprechende Fragen. Durch die ausschließliche Nutzung ohnehin vorhandener Daten entstehen bei diesem umgekehrten Ansatz - insbesondere für die Entwickler in einem Softwareentwicklungs-Projekt - keine besonderen zusätzlichen Aufwände zur Metrik-Datenerfassung. Dieser ressourcenschonende Aspekt stellt den Hauptvorteil des Ansatzes gegenüber der GQM-Methode dar.

2.3 Change-Request-Systeme

Ein Change-Request-System ist ein in Softwareentwicklungs-Projekten verwendetes Software-Werkzeug. Es dient der Erfassung und Bearbeitung von Tickets, deren Inhalt Anforderungen, Änderungswünschen und Fehlerberichten zu der im Rahmen des Projekts zu entwickelnden Software beschreiben.

Tickets sind anhand einer eindeutigen Kennung (z.B. durch eine fortlaufende Nummer) identifizierbar. Weiter sind sie einer Abteilung im Unternehmen oder einem einzelnen Mitarbeiter zugewiesen, der für die Bearbeitung des Tickets verantwortlich ist. Sie folgen dabei einem Lebenszyklus, der mit der Erstellung, d.h. der Erfassung durch das Change-Request-System, beginnt und (in der Regel) mit dem Schließen des Tickets endet. Ein geschlossenes Ticket repräsentiert die abgeschlossene Bearbeitung des im Ticket beschriebenen Sachverhalts. Die Bearbeitung und ihr Abschluss findet im Allgemeinen ergebnisoffen statt: Ein Ticket, das einen Fehlerbericht enthält, kann z.B. durch die Behebung des Fehlers geschlossen werden. Eine anderer möglicher Abschluss besteht darin, dass der berichtete Fehler nicht nachvollziehbar sei, woraufhin das Ticket ebenfalls geschlossen wird. Ein geschlossenes Ticket ist nicht zwingend unveränderlich. Wurde der beschriebene Sachverhalt irrtümlich als abgeschlossen betrachtet, kann ein geschlossenes Ticket wieder geöffnet werden und weitere Schritte im Workflow des Change-Request-Systems vollziehen.

Neben Informationen über den Sachverhalt, den Bearbeiter und den Bearbeitungsstatus besitzen Tickets weitere Eigenschaften, die sich aus dem Kontext ergeben, in dem das Change-Request-System verwendet wird. Ein zur Behebung von Software-Fehlern verwendetes Change-Request-System nutzt etwa eine Eigenschaft zur Erfassung der Schwere eines Fehlers. Ein Ticket aus einem System zur Anforderungsverwaltung hingegen besitzt z.B. eine Eigenschaft, das die Art der Anforderung beschreibt (Funktionale Anforderung, Randbedingung, Qualitätsanforderung).

Diese in den Tickets der Change-Request-Systeme enthaltenen Daten beschreiben im Softwareentwicklungs-Prozess durchgeführte Aktivitäten, und beinhalten Kennwerte wie die Bearbeitungsdauer und den Bearbeitungserfolg. Die Benutzeroberflächen und Programmschnittstellen von Change-Request-Systemen sind jedoch in der Regel auf die Kernfunktionalität (Erfassung und Bearbeitung der Tickets) des Systems fokussiert. Sie zeigen etwa einem Mitarbeiter die ihm zugewiesenen Tickets oder benachrichtigen einen Kunden über die Behebung eines von ihm gemeldeten Fehlers.

Im Rahmen dieser Diplomarbeit werden Change-Request-Systeme als Datenquellen genutzt, um die in ihnen enthaltenen Daten zur Analyse und Optimierung von Softwareentwicklungs-Prozessen weiter zu verwenden.

2.4 Sankey Diagramme

“Ein Sankey-Diagramm ist eine graphische[sic!] Darstellung von Mengenflüssen.”[4]. Die Mengenflüsse werden durch Pfeile dargestellt, deren Dicke proportional die transportierte Menge widerspiegelt. Der Diagrammtyp ist nach dem irischen Bauingenieur Henry Phineas Riall Sankey benannt, der ihn im Jahr 1898 zur “Verdeutlichung der Energie-

[4] `http://de.wikipedia.org/wiki/Sankey-Diagramm`, abgerufen am 10.06.2013

effizienz von Dampfmaschinen verwendete"[Sch06]. Anschließend wurde der Diagrammtyp aufgrund seiner intuitiven Verständlichkeit häufig zur Darstellung von Mengenflüssen verwendet. Das Sankey-Diagramm wurde allerdings bislang nie exakt definiert. Die praktische Verwendung hat viele verschiedene Varianten hervorgebracht. Heute finden sie vor allem bei der Visualisierung von Energieflüssen Verwendung (Abbildung 2.3). M. Schmidt beschreibt die historische Entwicklung der Sankey-Diagramme und gibt einen Überblick über ihre Varianten[Sch06]. Er leitet folgende gemeinsame Kerneigenschaften für Sankey-Diagrammen her:

- Die Mengengrößen sind extensive Größen, d.h. insbesondere: Sie sind addierbar.
- Die Pfeilbreite ist proportional zur dargestellten Menge.
- Die dargestellten Mengengrößen sind auf eine Zeitperiode bezogen.
- Es werden keine Bestandsgrößen berücksichtigt, d.h. es gibt keine Lagerbildung.
- Es wird stillschweigend von einer Energie- oder Masse-Erhaltung ausgegangen.

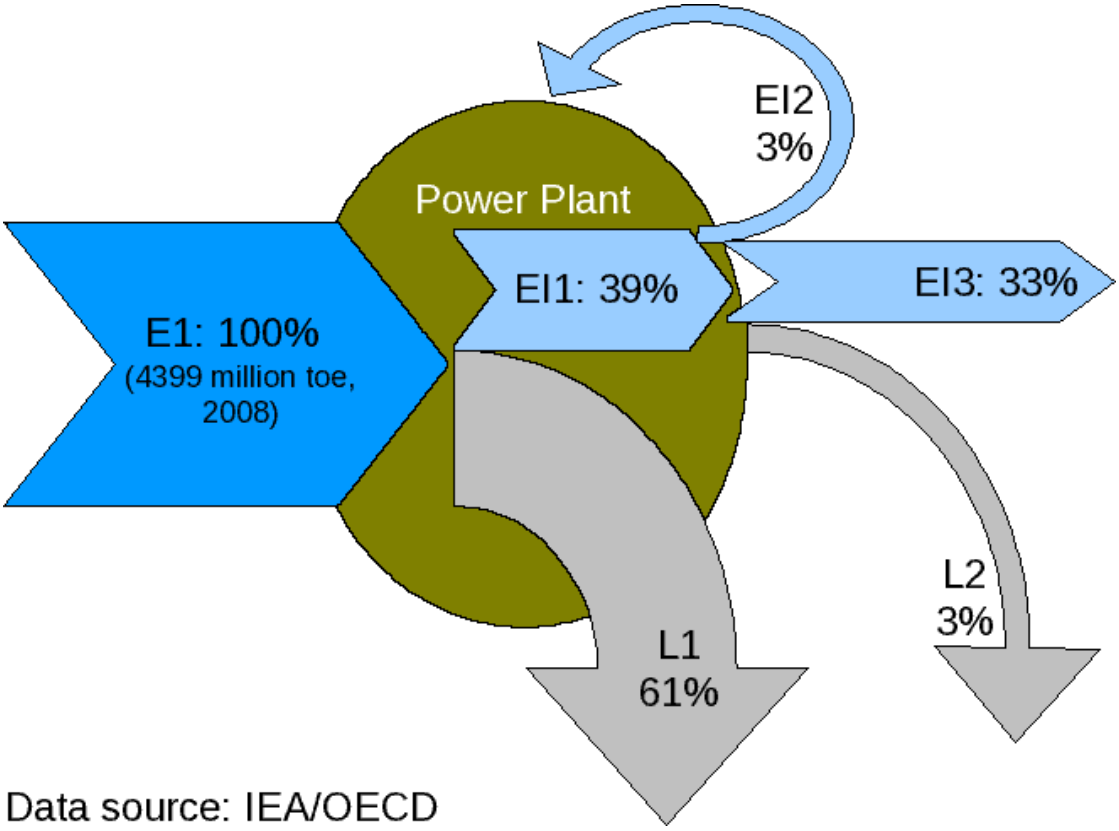

Abbildung 2.3: Energiefluss-Sankey-Diagramm
Quelle: `http://de.wikipedia.org/wiki/Sankey-Diagramm`, abgerufen am 10.06.2013

Anwendung in *River*

Die Historien der Tickets eines Change-Request-Systems lassen sich zu Gruppen von Tickets mit gemeinsamen Historien bzw. Historienfragmenten zusammenfassen. Sie bilden dabei Ticketflüsse, anhand derer sich die Eigenschafts-Entwicklung der Ticketgruppen darstellen lässt. Diese Gruppen werden im in dieser Diplomarbeit entwickelten Werkzeug *River*[5] über Sankey-Diagramme visualisiert.

Technisch wird dazu auf das Sankey-Plugin[6] der Javascript Bibliothek Data-Driven-Documents[7] zurückgegriffen. Dieses Plugin berechnet aus der JSON-Darstellung eines gerichteten, gewichteten und zyklenfreien Graphen eine Visualisierung in Form eines Sankey-Diagramms. Es ordnet dazu die Knoten des Graphen spaltenweise an. Knoten ohne eingehende Kanten werden links in der ersten Spalte und Knoten ohne ausgehende Kanten rechts in der letzten Spalte positioniert. Alle weiteren Knoten werden derart angeordnet, dass die Quellknoten aller eingehenden Kanten links des Knotens und die Zielknoten aller ausgehenden Kanten rechts des Knotens liegen. Der Layout-Algorithmus erzeugt dazu solange neue Spalten, bis diese Bedingungen erfüllt sind. Ein Knoten stellt einen Eigenschaftswert eines Tickets dar, wobei die Höhe des Knotens die Anzahl der Tickets mit diesem Eigenschaftswert repräsentiert. Die Kanten zwischen zwei Knoten zeigen die Änderung des Eigenschaftswerts einer Gruppe von Tickets. Die Breite der Kante symbolisiert die Anzahl der Tickets in dieser Gruppe. Quelltext 5.5 (S. 67) zeigt ein Beispiel der JSON-Darstellung eines Graphen und Abbildung 5.8 das Sankey-Diagramm, das das Sankey-Plugin daraus generiert.

Modifikation des Sankey-Plugins

Die Funktionalität des Sankey-Plugins wurde um folgende Merkmale erweitert, um Anforderungen des Kontextes, in dem es verwendet wurde, zu erfüllen:

- Im zweiten Prototyp wurden in das Sankey-Diagramm “hinein-fließende” Transitionen verwendet. Um eine solche Transition darzustellen, wird in der zugrundeliegenden JSON-Darstellung die entsprechende Kante ohne Angabe eines Quellknoten notiert.

- Die Transitionen haben einen “Pfeilkopf” erhalten, um die Richtung zu verdeutlichen.

- Es wurde ein Mechanismus zur Selektion von Graph-Elementen inklusive visuellem Feedback und Aufruf einer Callback-Methode im JSF-Backing-Bean implementiert.

- Das Plugin wurde modifiziert, um eine Aktualisierung des Sankey-Diagramms auch nach einem Ajax-Request und nicht nur nach einem vollständigen (Neu-)Laden der einbettenden Web-Oberfläche durchführen zu können.

[5] Der Name *River* (deutsch: Fluss) leitet sich von der Mengenfluss-Darstellung der Sankey-Diagramme ab, die innerhalb des Werkzeugs eine zentrale Rolle spielen.
[6] `http://bost.ocks.org/mike/sankey/`, abgerufen am 10.06.2013
[7] `http://d3js.org/`, abgerufen am 10.06.2013

2.5 Java EE

Dieser Abschnitt gibt einen Überblick über die Komponenten der Software-Architektur Java EE, die im Werkzeug *River* verwendet werden.

Java Server Faces (JSF)

"JavaServer Faces (JSF) ist ein Framework-Standard zur Entwicklung von grafischen Benutzeroberflächen für Webapplikationen."[8]. Die Definition einer Benutzeroberfläche in JSF geschieht nach dem Model-View-Controller-Konzept. Die View wird über eine oder mehrere XHTML-Dateien definiert, die aus JSF-Tags bestehen, die die GUI-Elemente deklarieren. Das Datenmodell wird durch Properties und zugehörige Getter und Setter in einer Backing Bean[9] realisiert. Die in der Backing Bean enthaltenen Methoden nehmen die Rolle des Controllers ein. Der Zugriff aus der View auf das Datenmodell und den Controller geschieht über Werte- und Methodenbindungen in Expression-Language (EL)-Ausdrücken.

River verwendet JSF zur Definition seiner Benutzeroberfläche und benutzt Komponenten aus der JSF-Komponenten-Bibliothek Primefaces[10].

Enterprise Java Beans (EJB)

Enterprise Java Beans (EJB) ist das Komponenten-Modell der Software-Architektur Java EE. Es ist dazu konzipiert, Lösungen zu verschiedenen in vielen Softwareprojekten wiederkehrenden Standard-Problemstellungen zur Verfügung zu stellen. Es erlaubt so den Entwicklern, sich stärker auf die spezifischen Problemstellungen des Software-Projekts zu konzentrieren. Zu den Standardproblemstellungen, die EJB adressiert, gehören Persistierung, Transaktionalität, Sicherheit und Nebenläufigkeit. EJB definiert drei verschiedene Bean-Varianten:

Session Beans beinhalten die Anwendungslogik einer Java EE Anwendung. *River* verwendet sie in der *stateless*-Variante. Diese Session Beans besitzen keinen eigenen Zustand, sondern basieren lediglich auf den durch den Aufrufer übergebenen Argumenten und dem über Entity Beans persistierten Daten-Modell der Anwendung.

Entity Beans dienen der Persistierung des Daten-Modells der Anwendung. Ein Entity-Manager kümmert sich um das Laden und Speichern der Entity Beans und abstrahiert von der zugrundeliegenden Datenspeichertechnologie (etwa einer Datenbank). *River* nutzt Entity-Beans zur Persistierung von Ticket-Daten und Ticket-Historien.

Message Driven Beans dienen der asynchronen Verarbeitung von JMS-Nachrichten in Java EE Anwendungen. *River* implementiert die *Ticket-Daten-Verarbeitung* über eine Message Driven Bean. Sie verarbeitet JMS-Nachrichten mit Ticket-Daten, die zuvor von *Adaptern* aus Change-Request-Systemen extrahiert wurden.

[8] `https://de.wikipedia.org/wiki/JavaServer_Faces`, abgerufen am 10.06.2013
[9] eine annotierte POJO (Plain Old Java Object)-Klasse
[10] `http://primefaces.org/`, abgerufen am 10.06.2013

Java Message Service (JMS)

Java Message Service ist ein Nachrichtenstandard über den Java EE Komponenten Nachrichten erzeugen, senden, empfangen und verarbeiten können. Der Standard garantiert die Zustellung der Nachrichten, und ermöglicht eine lose Kopplung zwischen den Kommunikationsteilnehmern, indem diese lediglich ein gemeinsames Nachrichtenformat verwenden müssen.

River definiert ein gemeinsames Nachrichtenformat über zwei Nachrichtentypen, die die Komponenten *initiale Ticket-Daten-Erfassung*, *Ticket-Daten-Aktualisierung* zur Übermittlung von Ticket-Daten an die *Ticket-Daten-Verarbeitung* des Werkzeugs verwenden. Diese Entkopplung der Komponenten erleichtert eine mögliche Integration und Wiederverwendung in andere Software-Projekte, insbesondere aus dem MeDIC Umfeld.

Für weitergehende Informationen zu Java EE sei der Leser auf die Arbeiten von E. Emelyanova[Eme12] und F. Evers[Eve12] verwiesen, die diese Software-Architektur detailliert beschreiben.

2.6 Weitere Technologien

Neben Technologien aus der Java EE Software-Architektur werden in dieser Diplomarbeit auch der XMLRPC Standard und das JSON Datenformat eingesetzt, die im Folgenden beschrieben werden.

Extensible Markup Language Remote Procedure Call (XMLRPC)

XMLRPC ist ein Standard, der den Methodenaufruf zwischen verteilten Systemen definiert und aufgrund seiner simplen Struktur von vielen Programmiersprachen und Softwaresystemen unterstützt wird[11]. Ein XMLRPC-Dienst bietet über das HTTP-Protokoll Methoden an, die ein XMLRPC-Klient durch eine HTTP-Anfrage aufrufen kann. Den Namen der aufzurufenden Methode und die zugehörigen Argumente überträgt der Aufrufer in einer XML-Darstellung im Body der HTTP-Anfrage. Nach der Ausführung der Methode erhält der Aufrufer Rückgabewerte oder Fehlermeldungen über die HTTP-Antwort. Quelltext 2.1 zeigt ein Beispiel des HTTP-Body für den Abruf einer Liste von Ticket-Ids über den XMLRPC-Dienst eines Trac-Systems.

```
1  <?xml version="1.0"?>
2
3  <methodCall>
4    <methodName>ticket.query</methodName>
5    <params>
6      <param>
7        <string>status!=assi</string>
8      </param>
9    </params>
10 </methodCall>
```

Quelltext 2.1: Body einer HTTP-Anfrage an einen XMLRPC-Dienst

[11] http://de.wikipedia.org/wiki/XML-RPC, abgerufen am 10.06.2013

In dem in dieser Diplomarbeit entwickelten Werkzeug *River* wird XMLRPC zum Zugriff auf die XMLRPC-Dienste der Change-Request-Systeme Trac und Redmine genutzt. Das Werkzeug agiert dabei als XMLRPC-Klient. Des Weiteren bietet *River* auch einen XMLRPC-Dienst an, über den Change-Request-Systeme ereignisgesteuert Ticket-Daten an das Werkzeug übertragen können.

JavaScript Object Notation (JSON)

JSON ist ein kompaktes und leicht lesbares Datenformat zum Datenaustausch zwischen verschiedenen Anwendungen oder Anwendungskomponenten. JSON unterstützt die Angabe einfacher Datentypen, wie Zeichenketten, Zahlen und boolschen Werten. Diese können mit Hilfe von Arrays (sortierte Liste von JSON-Elementen) und Objects (Menge von Schlüssel-Wert-Paaren) zu komplexeren Datentypen kombiniert werden, die wiederum als Elemente eines Arrays oder Objects verwendet werden können.

River setzt JSON ein, um das Graph-Modell eines Ticket-Eigenschafts-Flusses dem zu dessen Visualisierung genutzten Data-Driven-Documents-Sankey-Plugin (siehe Abschnitt 2.4) zur Verfügung zu stellen. Die JSON-Darstellung eines solchen Graph-Modells zeigt Quelltext 5.5.

3 Verwandte Arbeiten

Inhaltsangabe

Im Zentrum dieser Diplomarbeit steht die Extraktion vorhandener Daten aus in Softwareentwicklungs-Prozessen genutzten Software-Werkzeugen sowie die Aufbereitung und Visualisierung dieser Daten in einer zur Unterstützung der Analyse des Softwareentwicklungs-Prozesses geeigneten Form. Dieses Kapitel gibt einen Überblick über wissenschaftliche Arbeiten, die sich mit einem dieser beiden Aspekte befassen.

3.1 Extraktion und Analyse vorhandener Daten aus Software-Werkzeugen

Cook, Votta und Wolf beschreiben eine Methode[CVW98] zur Analyse routinemäßig in einem Prozess erhobener Daten mit dem Ziel, diesen Prozess besser zu verstehen und zu optimieren.

Nach dieser Methode wird zunächst ein Verständnis der Organisation, insbesondere ihrer Ziele, und des verwendeten Prozesses aufgebaut. Im zweiten Schritt werden verfügbare, vorhandene Datenquellen identifiziert. Aus den zuvor ermittelten Zielen werden Ziel-Metriken abgeleitet, die den Grad der Erreichung des Ziels messen. Anschließend werden auf den vorhandenen Datenquellen basierende Metriken definiert. Mit Hilfe einer statistischen Analyse wird schließlich die Existenz oder das Fehlen einer Korrelation zwischen den auf den Datenquellen basierenden Metriken und den Ziel-Metriken nachgewiesen.

Die Autoren illustrieren die Methode anhand einer Fall-Studie: Ein Software-Hersteller erhält von seinen Kunden Anfragen und Fehlerberichte. Als Datenquelle dient das Change-Request-System zur Erfassung der Kunden-Anfragen und Fehlerberichte. Ein Ziel des Herstellers ist eine hohe Kundenzufriedenheit. Dazu sollen berichtete Software-Fehler durch die Erstellung einer Software-Aktualisierung behoben werden und anschließend aus Sicht des Kunden auch behoben sein. Der Hersteller möchte die Zahl der Fälle, in denen eine Software-Aktualisierung aus Sicht des Kunden den Fehler nicht behoben hat, reduzieren. Die Ziel-Metrik misst daher, wie viele Software-Fehler-bezogene Tickets zur Zufriedenheit des Kunden geschlossen wurden. Die Datenquellen-basierten Metriken verarbeiten Daten des Change-Request-Systems, z.B. die Bearbeitungsdauer, den Bearbeiter usw. Die statistische Analyse ergab u.a, dass die Kunden-zufriedenstellende Bearbeitung stark vom bearbeitenden Mitarbeiter abhängt und sich die Aktivitäten, die die verschiedenen Mitarbeiter zur Erstellung eine Software-Aktualisierung durchführen, deutlich voneinander unterscheiden.

Der ausschließliche Nutzung vorhandener Daten stellt laut Cook, Votta und Wolf einen wesentlichen Kosten-Vorteil ihrer Methode gegenüber Verfahren dar, die zur Datenerfassung eine Instrumentierung des Prozesses benötigen.

In dieser Diplomarbeit wird ein sehr ähnlicher Ansatz untersucht. Die verwendete Vorgehensweise (siehe Abschnitt 4.1) orientiert sich daher stark an der oben beschriebenen Methode. An die Stelle der Auswahl einer Ziel-Metrik und der statistischen Korrelations-Analyse tritt hier jedoch eine geeignete Visualisierung der Datenquellen-basierten Metriken und die Korrelation zu Eigenschaften des Softwareentwicklungs-Prozesses durch einen Prozess-Experten. Die Visualisierung unterstützt dabei die Bildung eines hohen Verständnisses für die dargestellten Vorgänge im Softwareentwicklungs-Prozess.

3.2 Metrik-Aufbereitung und -Visualisierung

Die Visualisierung von Metriken ist ein Schwerpunkt des Forschungsprojekts "Metric Definition, Integration and Configuration (MeDIC)"[Via12] der Forschungsgruppe Software Construction. Mit Hilfe des Werkzeugs MeDIC-Dashboard[Han12] werden Metrik-Daten erfasst, aufbereitet und visualisiert. Im Rahmen von MeDIC sind einige studentische Abschlussarbeiten entstanden, die Fragestellungen behandeln, die auch für diese Diplomarbeit von Bedeutung sind.

Die Master-Arbeit "Konzeptionelle Erweiterung von Projektdashboards für unerfahrene Anwender"[Eve12] von F. Evers resultierte in einer ersten Version von MeDIC-Dashboard. Insbesondere seine Überlegungen zur Visualisierung von Metriken, die auf Stephen Few's Buch "Information Dashboard Design"[Few06] basieren, haben den GUI-Entwurf des in dieser Diplomarbeit entstandenen Werkzeugs beeinflusst.

Die Diplomarbeit "Variabilität von Metriken und Dashboard-Items im Umfeld von MeDIC"[Mä13] von M. Mädler untersucht, wie Standard-Metriken an individuelle Bedürfnisse angepasst werden können. Manche Details einer Metrik werden konfigurierbar, etwa die dargestellte Zeitspanne (z.B. letzte Woche, letzte 3 Monate, erstes Quartal) einer Metrik. Die Konfigurierbarkeit von Metriken spielt u.a. bei der Anwendung der Filter zu Einschränkung der dargestellten Ticket-Daten im Werkzeug *River* eine Rolle.

Schließlich stellt A. Steffens mit seiner Diplomarbeit "Entwurf eines Architekturmodells zur Integration heterogener Systeme in MeDIC"[Ste13] die Grundlage für die Architektur des Werkzeugs *River* zur Verfügung.

4 Konzept

Inhaltsangabe

Wie in Abschnitt 1.1 beschrieben, ist die zentrale Idee dieser Diplomarbeit, bereits vorhandene Daten aus Change-Request-Systemen zu analysieren und visualisieren, um Softwareentwicklungs-Prozesse zu verbessern. In Kapitel 3 wurde deutlich gemacht, inwiefern dieser Ansatz bereits in anderen Arbeiten angegangen wurde. Dieses Kapitel beschreibt die Konzepte hinter dieser Arbeit und ihre Auswirkungen auf die zu erstellende Werkzeugunterstützung *River*.

Nach einer Beschreibung der Vorgehensweise dieser Arbeit folgen Überlegungen über die Interessen und Ziele der Stakeholder im Kontext der Softwareentwicklungs-Prozessverbesserung, sowie über die Idee, die Visualisierung eines Softwareentwicklungs-Prozesses analog zur Visualisierung eines industriellen Prozesses in einer Messwarte eines Chemiebetriebes zu gestalten. Daraus werden Anforderungen an die Werkzeugunterstützung erhoben, die anschließend iterativ in zwei Prototypen validiert, ergänzt und auch verworfen werden. Das Kapitel schließt mit einer Auflistung der konsolidierten Anforderungen für das endgültige Werkzeug.

4.1 Vorgehensweise

In diesem Abschnitt wird die Vorgehensweise beschrieben, nach der diese Diplomarbeit bearbeitet wurde. Sie orientiert sich stark an der Methodik, die Cook, Votta und Wolf zur Analyse und Auswertung von Daten bereits vorhandener Software-Systeme vorschlagen[CVW98]:

1. *Verstehen der Organisation, des Projekts oder des Prozesses*: Was sind die Charakteristika des zu untersuchenden Prozesses? Welche Personen innerhalb der Organisation interessieren sich für die Ergebnisse der Prozess-Analyse? Mit diesen Fragen beschäftigen sich die beiden folgenden Abschnitte: Abschnitt 4.2 und Abschnitt 4.3. Insbesondere tragen jedoch die Evaluierungen der beiden Prototypen und des finalen Werkzeugs zum Verständnis der dazustellenden Prozesse bei.

2. *Identifikation der möglichen Datenquellen*: Aus welchen im Prozess verwendeten Software-Systemen können die den Metriken zugrundeliegenden Daten erhoben werden? Dieser Frage widmet sich der gleichnamige Abschnitt 4.4.

3. *Identifikation der Metriken, die den Erfolg des Prozesses messen*: Metriken, die den Erfolg eines Prozesses messen oder vorhersagen können, sind von besonderem Interesse bei der Analyse und Optimierung von Prozessen. Gleichzeitig sind insbesondere Vorhersage-Metriken in der Regel nicht über direkte und triviale Messungen erhebbar. Stattdessen werden Hypothesen über ihre Relation zu berechenbaren Metriken aufgestellt und geprüft. Dieser Punkt in der Methodik von Cook et al. wird innerhalb dieser Arbeit nicht bearbeitet und ist ggf. Gegenstand einer Folgearbeit. Diese Diplomarbeit legt den Fokus auf die visuelle Präsentation der Metriken.

4. *Identifikation der aus den Datenquellen berechenbaren Metriken*: Wie können die erhobenen Daten weiterverarbeitet und zu Informationsquellen über den Zustand der Prozesse verdichtet werden?

5. *Extraktion der Daten und Durchführung der Analyse*: Dieser Punkt sieht die Entwicklung von Programmmodulen zur Datenerhebung aus den vorhanden Software-Systemen sowie von Algorithmen zur Berechnung der Metriken vor.

6. *Interpretation der Ergebnisse*: Welche Schlüsse können aus den berechneten und dargestellten Metriken gezogen werden? Inwiefern tragen sie zur Verbesserung des Prozesses bei?

Punkt 3 (Identifikation der Metriken, die den Erfolg des Prozesses messen) der Methodik von Cook et al. wird innerhalb dieser Arbeit nicht bearbeitet und ist ggf. Gegenstand einer Folgearbeit. Diese Diplomarbeit legt den Fokus auf die Punkte 4-6 der Methodik und setzt diese anhand der erstellten Prototypen und der Werkzeugunterstützung *River* sowie den zugehörigen Evaluationen um.

4.2 Stakeholder

Als Stakeholder, d.h. Personengruppen, die ein besonderes Interesse an der Optimierung des Softwareentwicklungs-Prozesses haben, wurden Softwareprozess-Manager und Projektleiter identifiziert. In diesem Abschnitt werden ihre jeweiligen Aufgabenbereiche und Ziele beschrieben, um daraus Anforderungen für die Werkzeugunterstützung *River* abzuleiten.

Softwareprozess-Manager

Die Aufgabe des Softwareprozess-Managers ist der Entwurf, die Einführung, die Überwachung und kontinuierliche Optimierung von projektübergreifenden und organisationsweit standardisierten Softwareentwicklungs-Prozessmodellen[1]. In konkreten Projekten unter-

[1] `http://www.heise.de/developer/artikel/Erfolgreiche-Einfuehrung-von-Business-Process-Management-BPM-1715608.html` abgerufen am 25.05.2013

stützt er bei der ggf. notwendigen Anpassung des Prozessmodells an die projektspezifischen Anforderungen. Um diese Aufgaben, insbesondere die Optimierung, zu erfüllen, benötigt er Sichten auf die Umsetzung der verwendeten Prozesse.

Projektleiter

Der Projektleiter hat die Aufgabe, ein Software-Projekt erfolgreich durchzuführen, d.h. die Projektziele im Rahmen des angesetzten Zeit- und Kostenbudgets zu erreichen. Dazu wird das Projekt gemäß des verwendeten Softwareentwicklungs-Prozesses bearbeitet. Treten in dem Prozess unerwartete Abläufe auf, die die erfolgreiche Durchführung des Projekts gefährden oder behindern, ist es die Aufgabe des Projektleiters, gegenzusteuern. Auch er benötigt dazu Informationen über die Umsetzung des gewählten Prozesses.

Die Ziele des Softwareprozess-Managers sind eher langfristiger/strategischer Natur: Erfahrungen aus der Umsetzung eines Prozesses in einem Projekt fließen in die Verbesserung des organisationsweit standardisierten Prozesses zurück. Zur Überprüfung der getroffenen Verbesserungsmaßnahmen eignen sich insbesondere Trend-Darstellungen der Prozess-Eigenschaften, die die frühere Umsetzung des Prozesses der jetzigen gegenüber stellt.

Das wichtigste Ziel des Projektleiters - der Erfolg des Projekts - ist im Vergleich dazu kurzfristig. Weitere Ziele, wie der Aufbau von Erfahrungen bei der Umsetzung von Projekten, um ein nachfolgendes Projekt besser zu leiten, sowie die Kontrolle und Beobachtung dieses Fortschritts, sind zwar auch wichtig, gemessen am Erfolg des aktuellen Projekts aber zweitrangig. Daher bieten sich initial für Projektleiter Darstellungen an, die den aktuellen Zustand des Softwareentwicklungs-Prozesses charakterisieren. Daraus kann er ggf. notwendige Steuerungsmaßnahmen ableiten. Im Anschluss daran sind, ähnlich wie beim Softwareprozess-Manager, Trend-Darstellungen interessant, um Auskunft über die Wirksamkeit der Maßnahmen zu erlangen.

4.3 Leitstand

In industriellen Anlagen wie etwa Chemieanlagen oder Kraftwerken werden zur Überwachung der dort ablaufenden Prozesse Leitstände verwendet. Diese SCADA (Supervisory-Control-And-Data-Acquisition)-Systeme greifen auf Sensoren (z.B. Temperatursensoren) zu, um den Zustand des Prozesses zu messen. Ihre Daten werden von einem Prozessüberwachungssystem erfasst und als Kontrollvariable auf einer Anzeige oder einem Bildschirm des Leitstandes visualisiert. Zusätzlich findet meist eine automatische Überwachung der Kontrollvariablen statt. Dazu wird der Wertebereich der Kontrollvariablen in vier Kategorien unterteilt:

Normal

Die Kontrollvariable befindet sich innerhalb ihres Soll-Wertebereichs. Es sind keine Korrekturmaßnahmen erforderlich.

1. Warnstufe

Die Kontrollvariable befindet sich leicht außerhalb ihres Soll-Wertebereichs. Durch Regeln definierte Automatismen (wie z.B. das Ausschalten einer Heizung bei Überschreiten einer Höchsttemperatur) dienen dazu, die Variable wieder in die Kategorie “Normal” zu überführen. Alternativ greift ein Bediener in den Prozess ein. Es ist noch kein Schaden an der Anlage oder dem Produkt entstanden.

2. Warnstufe

Die Kontrollvariable weicht stark von ihrem Soll-Wertebereich ab. Schaden an der Anlage oder dem Produkt steht unmittelbar bevor. Hier greifen in der Regel automatische Notfall-Korrekturmaßnahmen, um fehlendes oder fehlerhaftes menschliches Eingreifen zu kompensieren, Schaden zu vermeiden und den Prozess wieder in einen sicheren Zustand zu führen.

Schaden entstanden

Die Kontrollvariable hat die 2. Warnstufe zeitlich dauerhaft überschritten und die Korrekturmaßnahmen waren erfolglos. Es ist Schaden an der Anlage oder dem Produkt entstanden.

Außerdem ermöglichen es SCADA-Systeme dem Bediener, steuernd in den Prozess einzugreifen. Er kann, wenn erlaubt, z.B. Grenzwerte für Kontrollvariablen festlegen oder Elemente des kontrollierten Systems wie etwa eine Pumpe manuell ein- und ausschalten.

Ein wesentliches Merkmal eines Leitstands ist die Zentralisierung der Überwachung und Steuerung. Er ersetzt bzw. ergänzt die Anzeigen vor Ort (etwa die Temperatur- und Druckanzeigen direkt an einem Kessel) und ermöglicht die ferngesteuerte Einflussnahme auf den Prozess. Muss der Messwart in den Prozess eingreifen, werden so Latenzzeiten reduziert bzw. vermieden, die z.B. entstehen würden, wenn er manuell ein Ventil öffnen oder schließen müsste.

Der Leitstand stellt die Elemente und Arbeitsschritte des Prozesses ikonisch dar und setzt sie logisch, zeitlich und/oder örtlich zueinander in Beziehung. So wird es dem Benutzer, der mit dem Prozess vertraut ist, erleichtert zu erkennen, welches Element der Visualisierung welchem Element des Prozesses entspricht.

Abbildung 4.1 zeigt ein typisches SCADA-System am Beispiel eines industriellen Prozesses. Die Darstellung ist darauf ausgelegt, den realen Prozess in einem Fließbild mit Prozessvariablen wiedererkennbar und intuitiv abzubilden. Selbst ein Laie erkennt schnell, ohne ein Handbuch konsultieren zu müssen, dass in der Mitte der Darstellung die Durchschnittstemperaturen einer Flüssigkeit in einem Kessel in 3 verschiedenen Höhen angezeigt werden. Daneben befinden sich die eingestellten Grenzwerte für das Ein- und Ausschalten der rechts dargestellten Wärmepumpe. Unten im Bild gibt es Knöpfe zur Konfiguration der Stellgrößen des Prozesses und zur Navigation zu Detailansichten sowie zur Visualisierung anderer Teile des Prozesses. Einen Prozess-Experten versetzt eine solche Darstellung in die Lage, auf einen Blick zu erkennen, ob sich der Prozess wie gewünscht verhält, oder ob und welche Korrektur-Maßnahmen erforderlich sind.

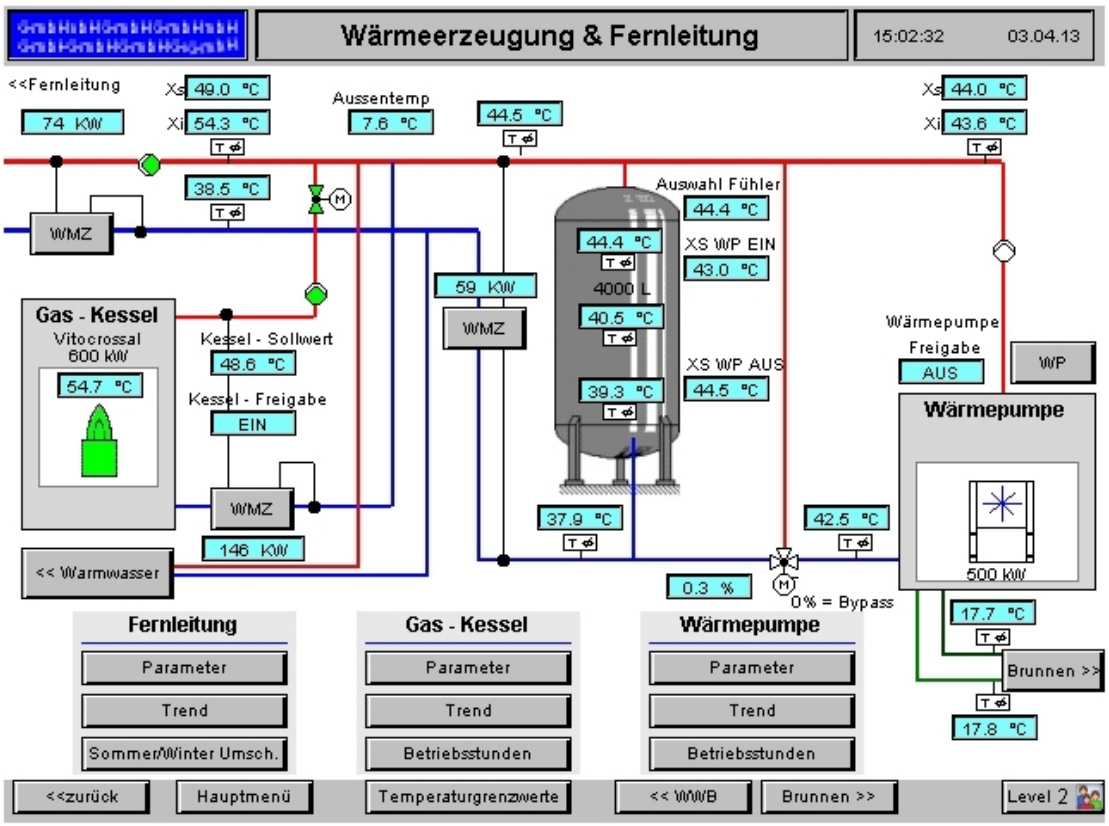

Abbildung 4.1: Schaltzentrale eines industriellen Prozesses
Quelle: `http://www.heise.de/security/meldung/Kritische-Schwachstelle-in-hunderten-Industrieanlagen-1854385.html`
abgerufen am 26.05.2013

Leitstände im Kontext von Softwareentwicklungs-Prozessen

Leitstände sind eine lange bewährtes Mittel zur Überwachung und Steuerung von industriellen Prozessen. Kann ein Softwareentwicklungs-Prozess auf ähnliche Art überwacht und gesteuert werden? Die Beantwortung dieser Frage erfordert ein genaues Verständnis, was eigentlich überwacht und gesteuert werden soll. Daher zunächst ein paar Begriffsdefinitionen:

Prozess: ein "sich über eine gewisse Zeit erstreckender Vorgang, bei dem etwas [allmählich] entsteht, sich herausbildet"[2].

Softwareentwicklungs-Prozess: The process by which user needs are translated into a software product.[ISO10]

Der Softwareentwicklungs-Prozess gliedert sich weiter auf in Phasen und Aktivitäten, an deren Ende jeweils Zwischenergebnisse stehen. Diese werden in späteren Phasen und Aktivitäten weiterverarbeitet, bis schließlich das Endprodukt entstanden ist.

Überwachung

Abbildung 4.2 zeigt einen Softwareentwicklungs-Prozess am Beispiel der Scrum-Methode[Abr02]. Die Aufteilung in Phasen ist durch senkrechte gestrichelte Linien gekennzeichnet. Die Aktivitäten befinden sich in rechteckigen Kästen und die Zwischenergebnisse sind durch Kästen mit geschwungenem unteren Rand dargestellt. Pfeile verdeutlichen die zeitliche Abfolge von Aktivitäten und stellen dar, welches Zwischenergebnis aus welcher Aktivität resultiert.

Diese Visualisierung eines Softwareentwicklungs-Prozesses ähnelt stark dem statischen Fließbild einer SCADA-Visualisierung. Es fehlt lediglich eine Anzeige von Kennwerten an und zwischen den angezeigten Prozess-Elementen, die Auskunft über den Zustand bzw. die Umsetzung der repräsentierten Phasen, Aktivitäten oder Zwischenergebnisse geben.

Ein Beispiel: Im Rahmen des "Sprint Planning Meetings" wird das "Sprint Backlog" mit denjenigen Funktionalitäten (in Form von "User-Stories") der zu entwickelnden Software befüllt, die im nächsten Sprint umgesetzt werden sollen. Diese Funktionalitäten sind jeweils mit einer Priorität und einer Aufwandsschätzung versehen. Der Prozess sieht vor, dass während des ein- bis vierwöchigen Sprints alle Entwickler gemeinsam an der Umsetzung der User-Story mit der höchsten Priorität arbeiten.

Um den Fortschritt und die Einhaltung des Prozesses während eines Sprints zu überwachen, könnte die Darstellung des Scrum-Prozesses um eine Angabe über den Zustand des "Sprint Backlogs" angereichert werden (siehe Abbildung 4.3). Die Darstellung erinnert an einen Kessel aus dem langsam Flüssigkeit abgelassen wird. Die User-Stories sind durch Abschnitte innerhalb dieser "Flüssigkeit" repräsentiert, wobei die Höhe eines solchen Abschnitts die Aufwandsabschätzung wiedergibt. Darüber hinaus sind sie nach Priorität sortiert und eingefärbt. Oben befindet sich die User-Story mit der höchsten (rot), unten die mit der niedrigsten Priorität (grün). Eine im Verlauf des Sprints von oben nach unten wandernde horizontale Linie zeigt den Fortschritt der Zeit an. Erledigte

[2] `http://www.duden.de/rechtschreibung/Prozess` abgerufen am 26.05.2013

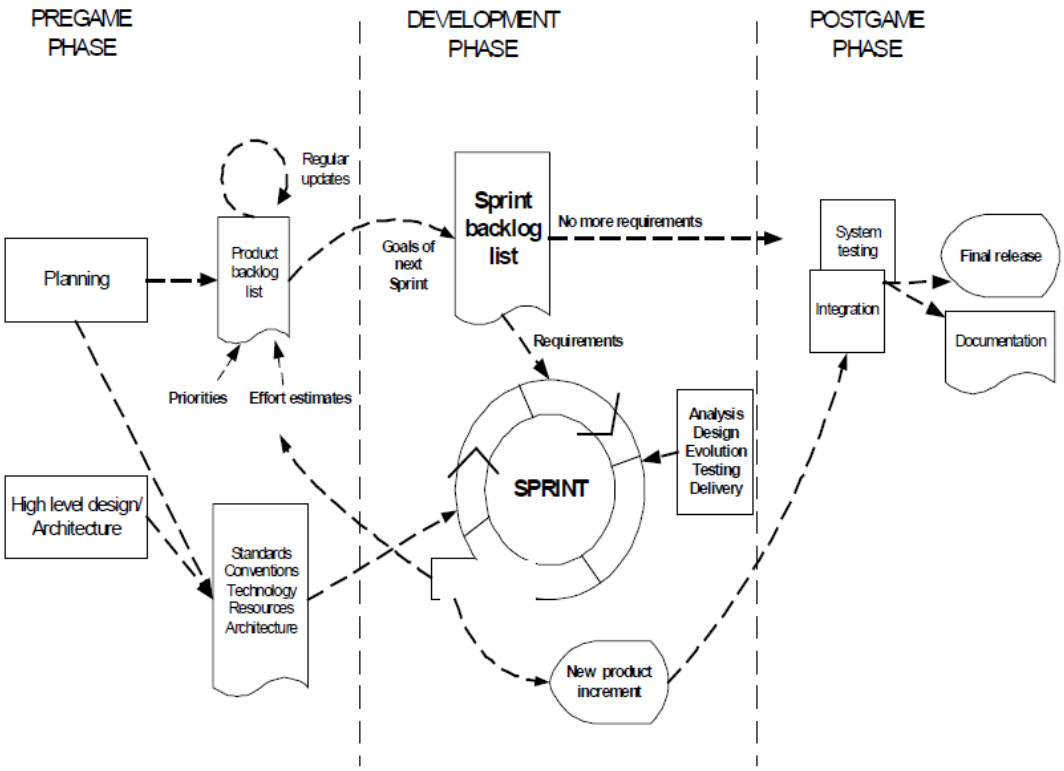

Abbildung 4.2: Der Scrum Prozess
Quelle: [Lic12]

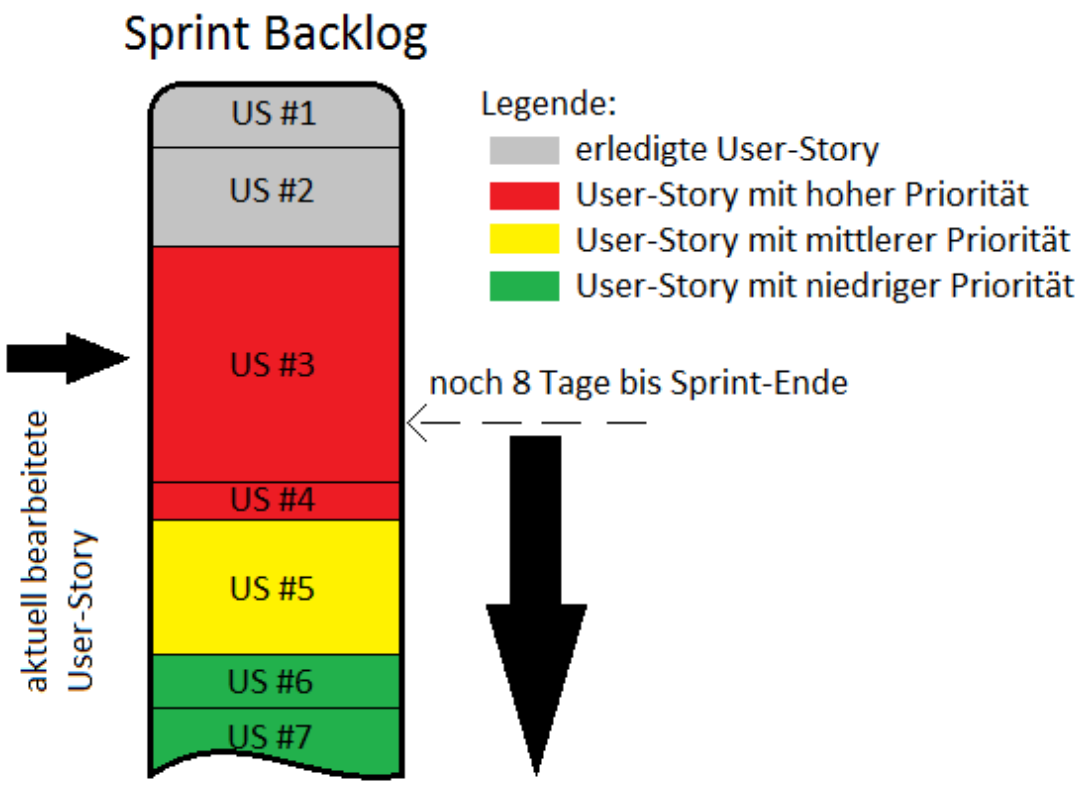

Abbildung 4.3: Backlog mit Kennwerten

User-Stories werden aus dem “Kessel” entfernt, indem sie grau dargestellt werden. Die aktuell bearbeitete User-Story ist mit einem Pfeil markiert.

Anhand dieser Visualisierung kann der Fortschritt des Sprints mit den Aufwandsabschätzungen der User-Stories auf einen Blick abgeglichen werden. Liegt der graue Teil der “Flüssigkeit” komplett über der Zeitlinie, und der farbige Teil komplett unter ihr, befindet sich der Sprint genau im Zeitplan. Bei einem farbigen Anteil oberhalb der Zeitlinie werden die User-Stories zu langsam abgearbeitet, und bei einem grauen Anteil unterhalb der Zeitlinie schneller als erwartet. Im Beispiel ist der Bearbeitungs-Fortschritt einer einzelnen User-Story nicht feingranular mit einem Prozentwert aufgeschlüsselt. Das Scrum-Team liegt hier also im Plan, sollte aber in Kürze die Bearbeitung von User-Story #3 fertigstellen.

Die Einhaltung des Prozesses ist ebenfalls erkennbar. Es sollte sich jeweils nur eine einzige User-Story (und nicht mehrere gleichzeitig) in Bearbeitung befinden, d.h. mit einem Pfeil markiert sein. Außerdem sollte es nur oben einen grauen Bereich geben, während der farbige Bereich unten liegt. Ist dies nicht der Fall und der farbige Bereich von grauen Anteilen durchsetzt, bedeutet dies, dass die User-Stories nicht entsprechend der im “Sprint Planning Meeting” festgelegten Priorität bearbeitet werden.

Steuerung

Dieses Beispiel zeigt, dass die Idee einer von industriellen Leitständen inspirierten Visualisierung und Überwachung von Softwareentwicklungs-Prozessen grundsätzlich umsetzbar ist. Wie aber sieht es mit der Steuerung des Entwicklungs-Prozesses aus? Können wie im obigen SCADA-Beispiel Knöpfe und Regler eingefügt werden, mit denen direkt Einfluss auf den Prozess genommen werden kann? Die Antwort lautet hier leider “Nein”. Im Gegensatz zu Prozessen in industriellen Anlagen sind die Maßnahmen, die als Reaktion einer Abweichung vom gewünschten Prozess-Verlauf zu treffen sind, im Kontext von Softwareentwicklungs-Prozessen in der Regel nicht technischer Natur und können somit auch nicht von einem technischen System umgesetzt werden. Was ist zu tun, wenn die obige Visualisierung des Scrum-Backlogs anzeigt, dass der Zeitplan im Sprint nicht eingehalten wird? Der “ScrumMaster” hat dann die Aufgabe, die Ursache der Prozessabweichung zu ermitteln und zu beseitigen. Eine mögliche Ursache wäre, dass die Aufwandsabschätzungen, die die Scrum-Teilnehmer den User-Stories zugeordnet haben, zu niedrig ausgefallen sind. In der Folge wurden zu viele User-Stories in das Sprint-Backlog übernommen, so dass es nicht gelingt, ihn innerhalb des Sprints vollständig abzuarbeiten. Mögliche Maßnahmen sind:

- Verlängerung des Sprints
- Verschiebung von User-Stories mit niedriger Priorität auf den nächsten Sprint
- Mehr Mitarbeiter zur Bewältigung der Aufgaben einsetzen (dies ist aber durch die kurze Dauer eines Sprints oft nicht zielführend. Stichwort: Einarbeitungszeit)
- Beim nächsten Sprint Planning Meeting besonderes Augenmerk auf die Aufwandsabschätzungen legen

Offensichtlich können diese Maßnahmen nicht einfach durch das Setzen einer technischen Stellgröße bewerkstelligt werden. Stattdessen trifft der ScrumMaster gemeinsam

mit dem Kunden, dem Product Owner und/oder den Scrum-Teilnehmern Entscheidungen bzw. weist auf die Prozessabweichung hin, um dann gemeinsam gegenzusteuern.

Zusammenfassung

Aus den obigen Überlegungen, wie Leitstände als Vorbild für die Analyse und Optimierung von Softwareentwicklungs-Prozessen genutzt werden können, ergibt sich der Fokus dieser Diplomarbeit: Der Überwachungs- bzw. Analyseanteil eines Leitstands lässt sich gut auf Softwareentwicklungs-Prozesse übertragen. Insbesondere eignet er sich zur Realisierung und Evaluierung einer Werkzeugunterstützung.

Der Steuerungsanteil hingegen findet vor allem über werkzeugferne Kanäle, wie Kommunikation unter den Mitarbeiten und organisatorischen Maßnahmen, statt. Eine gewisse Integration in ein Werkzeug ist zwar möglich, wie etwa die Benachrichtigung von Stakeholdern bei wichtigen Ereignissen im Prozess. Der wesentliche Anteil der Steuerung ist aber nicht automatisierbar. Daher wird der Steuerungsanteil im weiteren Verlauf dieser Diplomarbeit ausgeblendet und nicht weiter betrachtet.

4.4 Identifikation der Datenquellen

Dieser Abschnitt stellt die verschiedenen Datenquellen vor, die als Grundlage für die Analyse von Softwareentwicklungs-Prozessen in Frage kommen. Aufgrund der initialen Annahme dieser Diplomarbeit, dass sich die Prozesse in den verwendeten Werkzeugen und den von ihnen erzeugten Daten widerspiegeln, liegt der Fokus genau auf denjenigen Werkzeugen, die in praktisch jedem Softwareentwicklungsprojekt verwendet werden. Diese werden im Folgenden beschrieben und auf ihre Eignung als Datenquelle untersucht.

Zeit- und Kostenerfassungssysteme

In der Zeit- und Kostenerfassung notieren Mitarbeiter den von ihnen erbrachten Zeitaufwand und entstandene Kosten und ordnen sie Tätigkeiten und Arbeitspaketen zu. Auf Basis dieser Daten lassen sich leicht Aussagen über den Prozess treffen. In welchen Phasen und Aktivitäten entstehen besonders hohe Kosten? Wo weicht der tatsächliche Zeitaufwand von der Planung ab? Dieser Themenbereich ist allerdings bereits ausgiebig erforscht und mit Werkzeugunterstützung versehen. Als Beispiel sei hier die Earned-Value-Analyse[Lic12] genannt, die etwa mit dem verbreiteten Projektmanagement-Werkzeug MS-Project durchgeführt werden kann[3].

Versionsverwaltung

“Eine Versionsverwaltung ist ein System, das zur Erfassung von Änderungen an Dokumenten oder Dateien verwendet wird. Alle Versionen werden in einem Archiv mit Zeitstempel und Benutzerkennung gesichert und können später wiederhergestellt werden.”[4] Oft sind einer Änderung auch weitere Informationen zugeordnet. Dies kann eine Commit-Nachricht in Form eines Freitextes sein, die eine unstrukturierte Beschreibung

[3] `http://office.microsoft.com/en-us/project-help/analyze-project-performance-with-earned-value-analysis-HA101819808.aspx` abgerufen am 27.05.2013

[4] `https://de.wikipedia.org/wiki/Versionsverwaltung` abgerufen am 27.05.2013

der Änderung darstellt. Aber auch strukturierte Informationen wie die Zuordnung der Änderung zu einem bearbeiteten Änderungswunsch bzw. behobenen Fehler sind üblich. Anhand dieser Daten können vor allem Metriken erstellt werden, die die Software selbst vermessen. Z.B. könnte eine Metrik darstellen, in welchen Dateien, Modulen oder Komponenten besonders viele Fehler behoben wurden. Erfahrungsgemäß gilt für Softwarefehler das Pareto-Prinzip: Insbesondere dort wo viele Fehler gefunden wurden, die Fehlerdichte also hoch ist, sind weitere Fehler zu erwarten[CJ02]. Eine solche Metrik liefert also Hinweise, welche Teile der Software besonders intensiv getestet oder mit Reviews geprüft werden sollten.

Bezüglich des Softwareentwicklungs-Prozesses scheint die Aussagekraft der in Versionsverwaltungen enthaltenen Daten allerdings begrenzt zu sein. Welche Aussage kann über den Prozess getroffen werden, wenn die Fehlerdichte in einer Komponente niedrig ist? Zwei gegensätzliche Interpretationen liegen nahe:

1. Einerseits könnten die Fehlervermeidungsmaßnahmen, die der Prozess vorsieht, die Fehlerdichte erfolgreich niedrig gehalten haben. So könnte beispielsweise das Anforderungsdokument eine hohe Qualität besitzen und klar definierte Schnittstellen und Funktionalitätsbeschreibungen enthalten, wodurch Missverständnisse und Fehlinterpretationen in Design und Implementierung vermieden wurden.

2. Eine andere Möglichkeit ist eine schlechte Testabdeckung und/oder ungenügende Reviews. Es existieren zwar Fehler, diese werden aber nicht gefunden.

Diese beiden Interpretationen betreffen unterschiedliche Aspekte des Prozesses und bewerten sie gegensätzlich. Der Zusammenhang zwischen den Metrikmesswerten und der Schlussfolgerung durch die Interpretation ist hier also fragwürdig.

Change-Request-Systeme

Change-Request-Systeme wurden bereits in Abschnitt 2.3 näher beschrieben. Die Daten in Change-Request-Systemen korrelieren stark mit den Aktivitäten eines Softwareentwicklungs-Prozesses. Am Beispiel von Scrum werden User-Stories weiter zu kleineren Aufgaben (Tasks) aufgespaltet, die wiederum in das Change-Request-System in Form von Tickets eingetragen werden. Die Bearbeitung dieser Tasks wird durch eine Serie von Aktivitäten im Prozess vorangetrieben, im Einzelnen sind dies Analyse, Design, Evolution, Testing und Delivery (vgl. Abbildung 4.2). Der Fortschritt der Bearbeitung dieser Tasks spiegelt sich direkt im Status der Tickets wider. Auf Basis dieser Daten kann etwa ein stockender Sprint schnell erkannt werden. Über eine Auswertung der zu einer Task gehörenden Aufwandsabschätzung und den Abgleich mit dem Lebenszyklus einer Task (Zeit zwischen dem Status “neu” bis zum Status “abgeschlossen”) kann z.B. eine Metrik definiert werden, die die Güte der Aufwandsabschätzungen bewertet. Aus den gleichen Daten kann etwa auch die Teamvelocity errechnet werden, eine Größe mit deren Hilfe die Anzahl und der Umfang der User-Stories, die ein Team innerhalb eines Sprints sinnvoll bearbeiten kann, vorhergesagt wird.

Die genannten Beispiele stellen Informationen über den Zustand und den Ablauf des Prozesses zur Verfügung bzw. stellen Daten zur Verfügung, die zur Optimierung der Prozessumsetzung nützlich sind. Insgesamt ist die Datenquelle Change-Request-Systeme

damit die vielversprechendste und dient als Grundlage für die Metriken, die im während dieser Diplomarbeit entstandenen Werkzeug *River* visualisiert werden.

4.5 Erster Prototyp

Ausgehend von den gesetzten Zielen der Arbeit, den Überlegungen zu industriellen Leitständen und den Zielen und Interessen der Stakeholder wurde mit einem ersten Prototyp untersucht, wie ein GUI für eine Werkzeugunterstützung bei der Analyse und Verbesserung von Softwareentwicklungs-Prozessen aussehen könnte. Darüber hinaus wurden mit dem Prototyp erste Experimente mit den Technologien Trac, Python und JavaEE unternommen. Ein dritter Zweck des Prototyps ist die Erkundung der Datenerhebung: Wie können Daten aus dem im Prozess verwendeten Change-Request-System extrahiert werden?

In diesem Abschnitt werden daher zunächst funktionale wie auch nicht-funktionale Anforderungen an den Prototyp entwickelt. Diese Anforderungen werden anschließend anhand des erstellten Prototyps hinsichtlich ihrer Zweckmäßigkeit und Umsetzbarkeit überprüft. Die Anforderungen sind zwecks Referenzierbarkeit jeweils mit einem Kategoriekürzel und einer fortlaufenden Nummer markiert. Das Kategoriekürzel “RB” zeichnet eine Randbedingung aus, die Einschränkungen für den Entwurf definiert. “Q” steht für Qualitätsanforderung. In dieser Diplomarbeit werden hierunter vor allem Anforderungen an die Benutzbarkeit des Werkzeugs gestellt. “RB” und “Q” bilden zusammen die nicht-funktionalen Anforderungen. Eine mit “FA” markierte Anforderung ist eine funktionale Anforderung. Sie definiert, was das System tun soll.

- (RB1) Um den Einsatz und die Evaluierung des Werkzeugs in existierenden Softwareentwicklungs-Projekten zu erleichtern und eine hohe Akzeptanz zu erreichen, soll das Werkzeug aus Sicht der Software-Entwickler eines Projekts “nicht-störend” sein (vgl. [Joh01]. D.h. es soll von den Entwicklern keinen über ihre normalen Tätigkeiten hinausgehenden zusätzlichen Aufwand erfordern, wie etwa die manuelle Eingabe von Informationen über ihre Aktivitäten im Rahmen des Softwareentwicklungs-Prozesses.

- (Q1) Die Einarbeitungszeit in das Werkzeug soll für mit dem Softwareentwicklungs-Prozess vertraute Personen kurz sein.

- (Q2) Der Konfigurationsaufwand vor und während der Benutzung des Werkzeugs durch die Stakeholder soll gering sein.

- (Q3) Die Visualisierung soll übersichtlich und leicht verständlich sein.

- (FA1) Das Werkzeug soll den Benutzer bei der Analyse des Softwareentwicklungs-Prozesses unterstützen. Dazu zeigt es Informationen über den Prozess-Zustand an und hilft, Abweichungen vom gewünschten Prozess zu erkennen und zu bewerten.

- (FA2) Das Werkzeug soll Daten aus Change-Request-Systemen erheben, weiterverarbeiten und visualisieren.

Die genannten Anforderungen sind noch sehr grobgranular. Es ist aber gerade Zweck und Ergebnis des explorativen Prototyps, diese zu verfeinern. Des Weiteren lassen die Anforderungen bzgl. ihrer Überprüfbarkeit zu wünschen übrig. Da es sich bei diesem Werkzeug aber nicht um ein Projekt handelt, an dessen Ende ein Kunde anhand überprüfbarer Eigenschaften über Abnahme oder Ablehnung entscheidet, wird dies hier in Kauf genommen. An die Stelle der Abnahme tritt die Evaluierung, die Aufschluss darüber gibt, welche Metriken des Werkzeugs von den Evaluierungs-Teilnehmern als zweckmäßig zur Prozessanalyse und -optimierung bewertet werden. Die Formulierung der Anforderungen dient primär als Grundlage für Designentscheidungen in den Prototypen und dem endgültigen Werkzeug.

Beschreibung des Prototyps

Im Folgenden werden die aus den Anforderungen gefolgerten Designentscheidungen zum ersten Prototyp erläutert. Eine Beschreibung der technischen Realisierung befindet sich im Anhang A.1.

Um der Anforderung (Q1) nachzukommen, soll die Visualisierung im ersten Prototyp den Grundsatz der Wiedererkennbarkeit des Prozesses erfüllen. Wie in Abschnitt 4.4 beschrieben, korrelieren die Status-Felder von Tickets in Change-Request-Systeme gut mit den Aktivitäten im Prozess. Im ersten Prototyp wurde daher der Workflow des Change-Request-Systems, in Form eines gerichteten Graphen, als Basis für die Visualisierung gewählt. Die Knoten in diesem Graph repräsentieren die verschiedenen Status des Workflows, die Kanten die möglichen Übergänge von einem Status in einen anderen. Abbildung 4.4 zeigt die Oberfläche der ersten und einzigen[5] Version des ersten Prototyps anhand des Standard-Workflow eines Trac Issue Tracking Systems.

Moderne Change-Request-Systeme wie Trac, Jira oder Redmine stellen einerseits Standard-Workflows zur Verfügung, bieten andererseits aber die Möglichkeit, eigene Workflows zu definieren und zu verwenden. Dies dient dazu, das Change-Request-System an den Prozess anzupassen, in dem es verwendet wird. Wegen (Q2) soll vor der ersten Verwendung des Prototypen mit einem angepassten Workflow nicht zunächst manuell ein Graph zu zeichnen sein. Stattdessen verwendet der Prototyp einen Auto-Layouting-Algorithmus, der den Workflow-Graph in der üblichen Darstellung $G = (V, E)$ entgegennimmt, und nach einer Layout-Strategie den Graphen zeichnet.

In einem weiteren Schritt (d.h. weiteren Versionen des ersten Prototyps) sollte ursprünglich[5] dieser Graph um Informationen angereichert werden, die den Zustand des durch das Change-Request-System widergespiegelten Prozesses charakterisieren. Bei den Knoten wurde dabei z.B. an Füllstände gedacht. Wie viele Tickets befinden sich in welchem Zustand? Welche Prioritätsverteilung haben diese Tickets? Eine Visualisierung, die diese Fragen beantwortet, könnte mit einer Darstellung der Knoten ähnlich wie in Abbildung 4.3 bewerkstelligt werden. Für die Kanten liegt eine Visualisierung der Durchflussrate nahe. Z.B. wird über die Dicke des Kantenpfeils dargestellt, wie viele Tickets pro Zeiteinheit von einem Status in einen anderen wechseln. Eine solche Darstellung, so die Überlegungen zur Interpretation des Graphen, könnte genutzt werden, um eine stockende Bearbeitung von Tickets zu entdecken oder sogar vorherzusagen. “Fließen” etwa

[5] Der erste Prototyp wurde nicht weiterentwickelt. Eine Erläuterung der Gründe dafür ist unter “Gewonnene Erkenntnisse” zu finden.

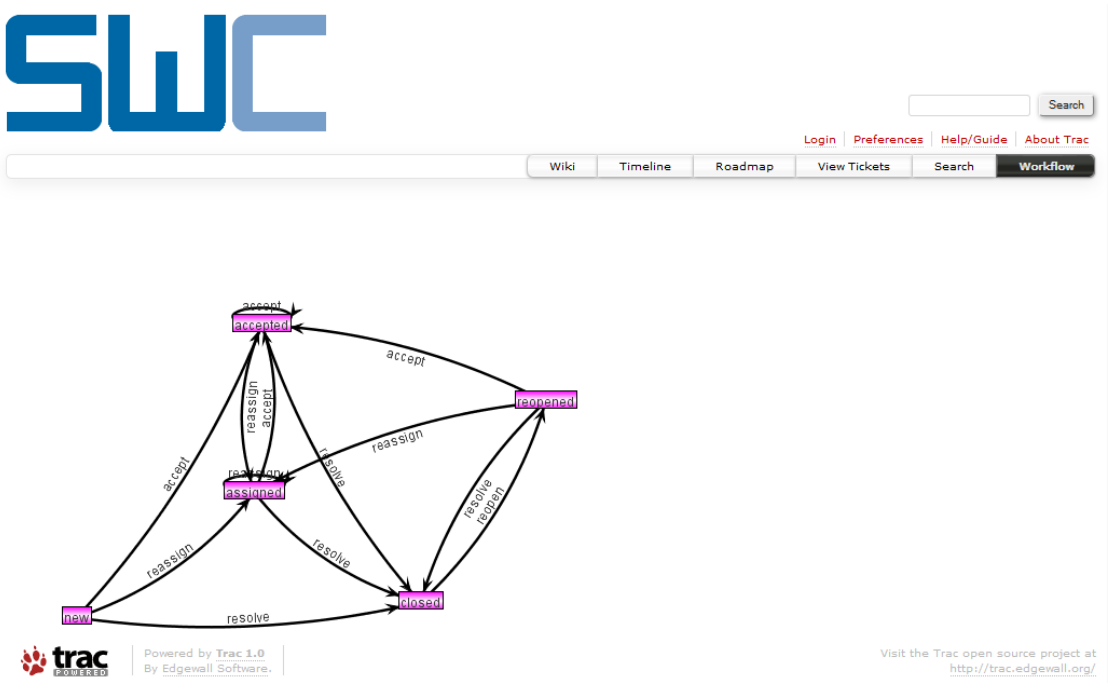

Abbildung 4.4: Erster Prototyp

über einen längeren Zeitraum mehr Tickets aus dem Status “assigned” in den Status “accepted” als von “accepted” zu “closed”, nehmen die Ticket-Bearbeiter mehr Tickets neu in Bearbeitung als sie fertigstellen. Es ist ggf. ein Anstieg der durchschnittlichen Ticket-Bearbeitungszeit zu erwarten. “Stauen” sich Tickets im Status “assigned” auf und “fließen” kaum weiter, wurden sie womöglich versehentlich einem in Urlaub befindlichen Mitarbeiter zugewiesen und wichtige Tätigkeiten bleiben unbemerkt unerledigt.

Neben Metriken, die direkt über die Darstellung des Knotens oder der Kante selbst visualisiert werden können (Füllhöhe, Pfeildicke), sollte es aber auch möglich sein, andere Metrikdarstellungen Knoten und Kanten zuzuordnen, z.B. in MeDIC-Dashboard genutzte Visualisierungen (siehe Abbildung 4.5). Liniendiagramme (etwa mit Füllstand oder Durchflussrate auf der y-Achse und der Zeit auf x-Achse) sind beispielsweise geeignet, um Trends darzustellen, wie sie insbesondere für Softwareprozess-Manager interessant sind. Histogramme setzen die durch den Knoten oder die Kante dargestellten Informationen in den Kontext eines weiteren Kriteriums, wie etwa die Ticketpriorität. Bulletgraphen helfen mit der Einordnung in Kategorien wie “gut”, “akzeptabel” und “schlecht” (Ampeldarstellung) und der Bereitstellung eines Vergleichswerts (das kann ein Sollwert sein, aber auch z.B. der Wert der vorherigen Messung) bei der Interpretation der dargestellten Metrik. M. Gora beschreibt diese und weitere Visualisierungen in seiner Bachelor-Arbeit[Gor13].

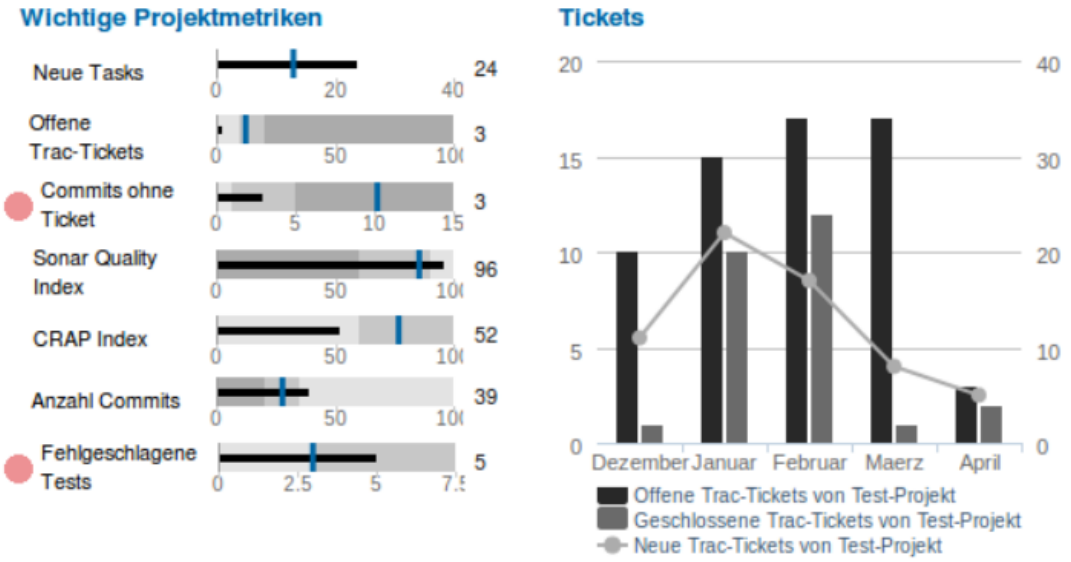

Abbildung 4.5: Bulletgraph (links) und kombiniertes Liniendiagramm/Histogramm (rechts) aus MeDIC-Dashboard
Quelle: [Gor13]

Gewonnene Erkenntnisse

Wie können diese zusätzlichen Metriken unter Beachtung von (Q3) (Übersichtlichkeit, Verständlichkeit) in den Workflow-Graph integriert werden? Zwei Bedingungen müssen dazu erfüllt sein:

1. Die Zuordnung muss eindeutig sein. Es muss auf eine Blick ersichtlich sein, welche Metrik zu welchem Graphelement (Knoten, Kante) gehört.

2. Die Metrik muss lesbar bleiben. Insbesondere darf die Skalierung der Metrik nicht zu klein werden, und die Metrik muss sich vollständig in einem freien Bereich zwischen den Graphelementen befinden. In keinem Fall darf etwa eine Graphkante mitten durch ein Liniendiagramm verlaufen.

Schon am obigen Beispiel des simplen Trac-Standard-Workflow ist erkennbar, dass die Erfüllung dieser Bedingungen keine triviale Aufgabe ist, und die gewählte Auto-Layout-Strategie dafür ungeeignet ist. Es ist nicht offensichtlich, wie etwa ein Liniendiagramm so in diesem Graphen untergebracht werden kann, sodass intuitiv erkennbar ist, dass es zur Kante von “accepted” nach “assigned” gehört und gleichzeitig gut lesbar bleibt.

Eine mögliche Lösung für dieses Problem stellen planare, orthogonale Graph-Layout-Algorithmen dar[ET94]. Diese wurden ursprünglich im Bereich der Very-Large-Scale-Integration (VLSI) dazu entwickelt, Transistoren in einem integrierten Schaltkreis optimal (nur senkrechte und waagerechte Leiterverbindungen, keine/wenig Biegungen, keine Leiterkreuzungen, kleinstmögliche Fläche) anzuordnen. Abbildung 4.6 zeigt den Standard-Trac-Workflow in einem orthogonalen, planaren Graphen. Die Knoten sind dabei an einem groben Raster bzw. Gitternetz ausgerichtet, während die Kanten über die Linien eines feineren Gitternetzes verlaufen (es ist feiner als das Knotenraster, um

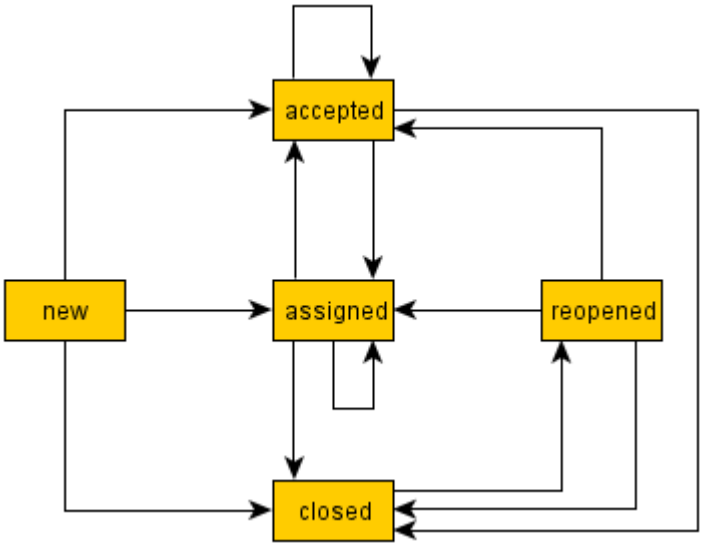

Abbildung 4.6: Standard Trac-Workflow in orthogonaler, planarer Darstellung

mehrere parallele verlaufende Kanten zu ermöglichen)[Fö97]. Um zusätzliche Metrik-Visualisierungen an den Kanten und Knoten anzubringen, könnte der Graph gestreckt werden, z.b. über die Vergrößerung des Abstandes zwischen den Gitternetzlinien, bis genügend Platz neben bzw. über oder unter den Graphelementen vorhanden ist, um einerseits die Metrik-Visualisierung in einem freien Bereich lesbar darzustellen, und sie andererseits so zu positionieren, dass durch ihre Nähe zu genau einem Graphelement intuitiv ersichtlich ist, auf welches Element sie bezogen ist. Es sind allerdings nicht alle Graphen planar oder planar darstellbar. Der vollständige Graph mit 5 Knoten K_5 ist ein solcher nicht planar darstellbarer Graph[6]. Durch die in modernen Change-Request-Systemen vorhandene Möglichkeit, Workflows frei zu definieren, kann nicht ausgeschlossen werden, dass hierbei nicht planare Graphen entstehen, wie etwa ein Workflow mit 5 Status, der den Übergang von jedem Status in jeden anderen erlaubt (und somit isomorph zu K_5 ist). In nicht planaren, orthogonalen Graphen, d.h. orthogonalen Graphen mit Kreuzungen, wird es aber wiederum schwieriger, geeignete Positionen für Metrik-Annotationen, die die beiden oben definierten Bedingungen erfüllen, zu berechnen.

In Absprache mit dem Betreuer der Diplomarbeit wurde an dieser Stelle die weitere Untersuchung dieser Visualisierung unterbrochen. Der Fokus dieser Arbeit ("Wie können in Softwareentwicklungs-Werkzeugen vorhandene Daten über Softwareentwicklungs-Prozesse ausgewertet und zur Prozessverbesserung verwendet werden?") sollte nicht aus den Augen verloren werden und sich nicht vorwiegend auf graphentheoretische Überlegungen verlagern. In einem zweiten Prototyp (siehe Abschnitt 4.7) sollte daher eine Visualisierung der Daten eines Change-Request-Systems gefunden werden, die sich "besser" zur Darstellung der Vorgänge im Softwareentwicklungs-Prozess eignet.

[6] `https://de.wikipedia.org/wiki/Planarer_Graph` abgerufen am 28.05.2013

4.6 Vom Workflow zum Sankey-Diagramm

Was ist mit der Forderung nach einer "besseren" Visualisierung am Ende des letzten Abschnitts genau gemeint? Einerseits eine technisch leichtere Umsetzung der Darstellung bei gleichzeitiger Beachtung der Anforderung nach Übersichtlichkeit und Verständlichkeit (Q3). Andererseits wurde im ersten Prototypen die Darstellung des Ticket-Workflows als Visualisierung hauptsächlich ausgewählt, um die Wiedererkennbarkeit des Softwareentwicklungs-Prozesses zu gewährleisten. In diesem Abschnitt wird die Frage untersucht, welche Arten von Informationen in einem Change-Request-System vorhanden sind und welche zum Workflow alternativen Darstellungsformen zu ihrer Visualisierung geeignet sind.

Im vorhergehenden Abschnitt wurden Annotationen von Change-Request-System-Workflows mit "Füllständen" an Knoten und Transitionsraten an Kanten skizziert. Diese beiden Aspekte ergeben sich gerade aus den typischen Daten, die Change-Request-Systeme produzieren. Sie halten einerseits die aktuellen Zustände der Tickets vor, aus denen der "Füllstand" resultiert. Anderseits stellt die Historie der Tickets die Datengrundlage für die Transitionsraten bereit.

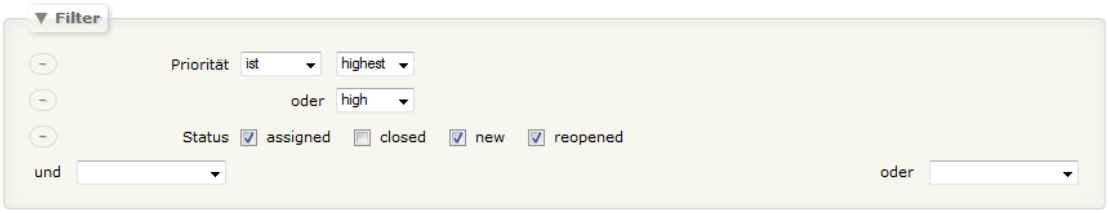

Abbildung 4.7: Eingabemaske zur Angabe eines Filters bei Ticketabfragen in Trac

Auswertungen über die aktuellen Ticket-Zustände unterstützen Change-Request-Systeme bereits über ihre Ticket-Abfrage-Mechanismen sehr gut. Der Anwender gibt dazu über einen Filter Kriterien vor, und erhält eine Liste aller Tickets, deren Eigenschaften die Kriterien erfüllen. Abbildung 4.7 zeigt die Oberfläche zur Eingabe eines Filters am Beispiel des Change-Request-Systems Trac. Nach einer Ticket-Abfrage mit diesem Filter erzeugt das System Ergebnisliste mit allen Tickets, deren Priorität "highest" oder "high" ist, und die sich in einem der Status "assigned", "new" oder "reopened" befinden. Über die Combo-Boxen "und" bzw. "oder" lässt sich der Filter leicht um zusätzliche Kriterien erweitern. Teil der Ergebnisliste ist selbstverständlich auch die Anzahl der zu den Kriterien passenden Tickets.

Im Vergleich zur Darstellung des Prozess-Workflows besitzt die Ergebnisliste einer Abfrage sowohl Vor- als auch Nachteile. Im Workflow ist die aktuelle Verteilung der Tickets gemäß des Ticketstatus "auf einen Blick" erkennbar. Das leistet die Ergebnisliste nicht, punktet dafür aber bei der Flexibilität. Über den Filter können Tickets nach einer beliebigen Kombination von Ticketeigenschaften ausgewählt werden. Im Workflow werden die Tickets fest nach dem Kriterium "Status" unterteilt, und ggf. mit Kontextinformationen über genau eine weitere Ticketeigenschaft versehen (vgl. Prioritäts-Schichten in Abbildung 4.3).

Die Historie der Tickets lässt sich hingegen mit Bordmitteln von Change-Request-Systemen derzeit (Stand: Mai 2013) überhaupt nicht (Trac, Redmine) oder nur rudi-

mentär (JIRA: Closed Vs Resolved Statistik) auswerten. Fragen wie: “Welche und wie viele Tickets mit hoher Priorität sind innerhalb der letzten Woche vom Status “in Bearbeitung” in den Status “im Test” übergegangen?”, oder: “Welche und wie viele Tickets haben den Status “im Test” in ihrem Lebenszyklus übersprungen und sind ungetestet geschlossen worden?”, oder: “Wie hoch ist der Anteil dieser Tickets an den allen Tickets?”, lassen sich aufgrund der fehlenden Bezugsmöglichkeit auf die Historie nicht über die verfügbaren Filterkriterien formulieren.

Eine Fokussierung auf die Visualisierung der Historie von Tickets ist also allein deshalb für den zweiten explorativen Prototyp schon interessant, weil sie von den Change-Request-Systemen bisher nicht bereitgestellt wird. Hier verbergen sich möglicherweise bislang nicht sichtbare Informationen, die die Stakeholder Softwareprozess-Manager und Projektleiter zur Einschätzung des Prozesszustands und zur Prozessverbesserung nutzen können.

Die Ticket-Historie beschreibt die Veränderungen der Ticketeigenschaften im Verlauf der Zeit. Dabei bilden mehrere Tickets, die in einer bestimmten Eigenschaft, wie etwa dem Ticketstatus, dieselbe Veränderung aufweisen, einen “Ticket-Fluss”: Die Tickets fließen von einem Status in einen anderen. Zur quantitativen Visualisierung dieser Flüsse eignen sich insbesondere, wie in Abschnitt 2.4 beschrieben, Sankey-Diagramme. Dieser Diagrammtyp wurde daher als zentrales Visualisierungselement für den zweiten Prototyp verwendet. Eine genaue Beschreibung des zweiten Prototyps sowie der Semantik eines Sankey-Diagramms im Kontext von Ticket-Flüssen befindet sich im folgenden Abschnitt 4.7.

4.7 Zweiter Prototyp

Das primäre Ziel des zweiten Prototypen ist die Evaluierung von Sankey-Diagrammen zur Darstellung von Ticket-Flüssen. Ein weiteres Ziel ist die technische Erkundung der Datenerhebung (Wie können Daten aus dem im Prozess verwendeten Change-Request-System extrahiert werden?).

Dieser Abschnitt beschreibt zunächst die Oberfläche des Prototyps, die Semantik des dargestellten Sankey-Diagramms und die getroffenen Design-Entscheidungen. Der fertige Prototyp wurde im Rahmen des “GDIS Info Cafe” Führungskräften der Generali Deutschland Informatik Services GmbH (GDIS) vorgestellt. Das dabei erhaltene Feedback wird am Ende dieses Abschnitts wiedergegeben. Die technische Realisierung des zweiten Prototyps ist im Anhang A.2 erläutert.

Beschreibung des Prototyps in Variante 1

Abbildung 4.8 zeigt die Oberfläche des Prototyps in Variante 1 (eine verbesserte Variante 2 wird weiter unten beschrieben) anhand eines beispielhaften Ticketflusses eines Trac-Change-Request-Systems, das den Trac-Standard-Workflow verwendet (vgl. Abbildung 4.4). Die Visualisierung stellt die Veränderung der Status der im Trac-System enthaltenen Tickets innerhalb eines Betrachtungs-Zeitraums in einem Sankey-Diagramm dar.

Das Diagramm wird von links nach rechts “gelesen”. Die Knoten sind dazu spaltenweise übereinander angeordnet. Es gibt eine erste Spalte (ganz links), deren Knoten im

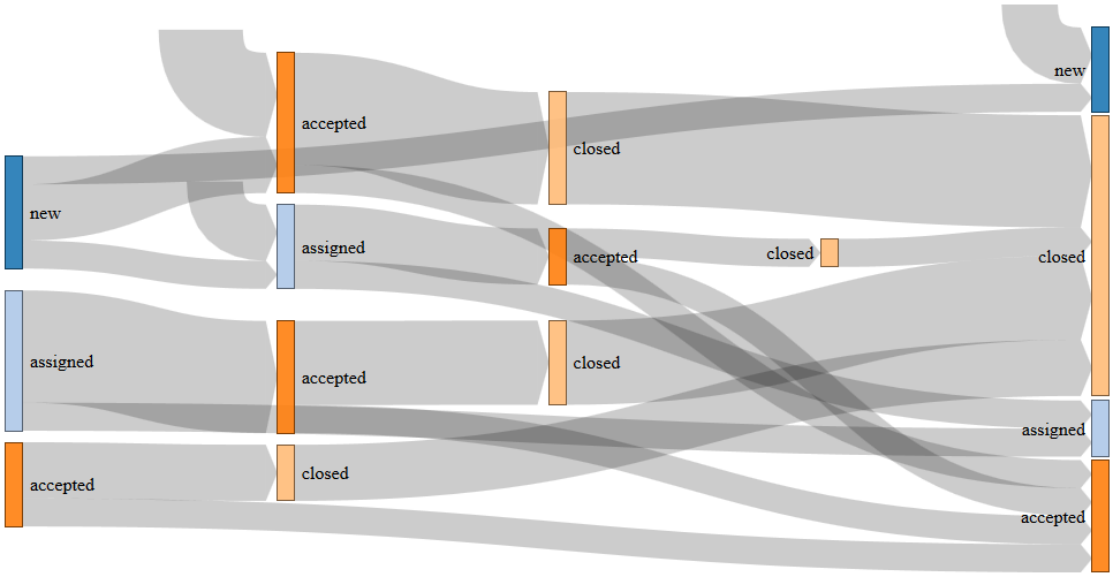

Abbildung 4.8: Variante 1 des zweiten Prototyps

Folgenden Startknoten genannt werden. Die Knoten der letzten Spalte (ganz rechts) heißen Endknoten. Beide Arten nehmen eine besondere Rolle gegenüber den Knoten (innere Knoten) der dazwischenliegenden Spalten ein. Die Startknoten in der ersten Spalte veranschaulichen die Verteilung der Ticket-Status zu Beginn des Betrachtungs-Zeitraums. Tickets mit gleichem Status sind dabei in einem Startknoten zusammengefasst. Die Anzahl der Tickets mit gleichem Status wird über die Höhe des entsprechenden Knotens visualisiert. In diesem Beispiel befinden sich zu Beginn des betrachteten Zeitraums 4 Tickets im Status “new”, 5 Tickets im Status “assigned” und 3 Tickets im Status “accepted” (insgesamt 12 Tickets). Die Endknoten in der letzten Spalte zeigen, wie viele Tickets welchen Status am Ende des Betrachtungszeitraums innehaben (hier: 3x“new”, 10x“closed”, 2x“assigned” und 4x“accepted”, insgesamt 19 Tickets). Die Visualisierung lässt zeigt zunächst nur die relative Verteilung der Tickets. Die absolute Zahl der durch Knoten (und Kanten) repräsentierten Tickets kann der Benutzer durch den Mouse-Over-Tooltip über dem entsprechenden Sankey-Diagramm-Element abfragen.

Die Startknoten, die Tickets mit gleichem Status zu Beginn des Betrachtungszeitraums zusammenfassen, fächern sich in nach rechts verlaufende Pfade auf. Ein solcher Pfad repräsentiert eine Gruppe von Tickets, die sich zu Beginn des Betrachtungs-Zeitraums im gleichen Status befanden und innerhalb des Betrachtungs-Zeitraums die gleiche Abfolge an Status-Änderungen durchlaufen haben. Jeder innere Knoten auf einem solchen Pfad steht somit anschaulich für Tickets mit identischer Status-Historie (seit Beginn des Betrachtungs-Zeitraums) und der Weg von einem Startknoten zu diesem inneren Knoten ist eindeutig. Nach einem inneren Knoten kann sich der Pfad weiter aufspalten, und so die weitere (unterschiedliche) Status-Entwicklung der durch den inneren Knoten repräsentierten Tickets darstellen.

Auf diese Weise bilden die Pfade, die den gleichen Startknoten besitzen, eine Art Korridor (gut erkennbar in Variante 2 des Prototyps, siehe Abbildung 4.9). Zwischen diesen Korridoren gibt es an den inneren Knoten keine Querverbindungen, d.h. Tickets mit un-

terschiedlichem Startknoten bzw. unterschiedlicher Statushistorie werden nicht wieder zu gemeinsamen Pfaden zusammengeführt, auch wenn sie etwa ab einem gemeinsamen Status identische restliche Statusveränderungen vollziehen. Erst die letzte Transition jedes Pfades führt zu einem nicht mehr nach Korridoren unterschiedenen Endknoten, der alle Tickets mit identischem Status zum Ende des Betrachtungs-Zeitraum unabhängig von ihrer Status-Historie konsolidiert.

Die jeweils letzte Transition jedes Pfades steht allerdings nicht mehr für eine tatsächliche Ticketstatus-Änderung. Sie dient lediglich der Zusammenführung von Tickets mit identischem Status zum Ende des Betrachtungs-Zeitraums. Die letzte Transition jedes Pfades verbindet daher auch ausschließlich gleich bezeichnete Knoten und kann durchaus auch die einzige Transition des Pfades sein, wie z.B. der Pfad mit nur einem Ticket vom Startknoten "new" links oben zum Endknoten "new" rechts oben. Die Semantik ist hierbei: Eines der vier Tickets, die zu Beginn des Betrachtungs-Zeitraums den Status "new" innehatten, hat im Laufe des Zeitraums keine Status-Änderung vollzogen und befindet sich zum Ende des Zeitraums nach wie vor im Status "new".

Neben den bisher beschriebenen Elementen des Sankey-Diagramms, die die Status-Änderungen existierender Tickets innerhalb des Betrachtungs-Zeitraums visualisieren, stellen die von oben "hinein fließenden" Transitionen Tickets dar, die während des Betrachtungs-Zeitraums neu erstellt wurden. Sie führen ausschließlich in die 2. Spalte im Korridor des Ticketstatus "new" (siehe links in Abbildung 4.8 oder an den Endknoten "new" (siehe rechts oben in Abbildung 4.8). Erstere repräsentieren neu erstellte Tickets, deren Status sich im Betrachtungs-Zeitraum ändert, letztere stehen für Tickets, die im Status "new" verbleiben. Die Sonderrolle des Status "new" ergibt sich daraus, dass sich alle Tickets bei ihrer Erstellung zunächst in diesem Status befinden.

Eine ähnliche Sonderrolle könnte auch der Status "closed" einnehmen. Der Lebenszyklus eines Tickets endet in der Regel mit dem Schließen des Tickets. Es kann dabei durchaus mehrere schließende Status geben, die z.b. nach Lösungsart differenziert sind. Einige Beispiele: "closed/resolved", "closed/cannot reproduce bug", "closed/won't fix". Es liegt nahe, in der Visualisierung die Transition zu einem "closed"-Knoten und den "closed"-Knoten selbst durch einen "herausfließenden" Pfeil zu substituieren. Zwei Diagramm-Elemente werden damit zu einem verschmolzen, ohne den Informationsgehalt zu reduzieren. Die Übersichtlichkeit der Darstellung verbessert sich. Allerdings zeichnen "closed"-Knoten nicht zwingend und eindeutig das Ende des Lebenszyklus eines Tickets aus, wie dies beim Status "new" und dem Beginn des Lebenszyklus der Fall ist. Der Trac-Standard-Workflow sieht beispielsweise vor, Tickets im Status "closed" über die Transition "reopen" wieder zu öffnen und in den Status "reopened" zu überführen. Die Visualisierung des Prototyps müsste daher Tickets, die im Betrachtungs-Zeitraum geschlossen werden und geschlossen bleiben, von Tickets differenzieren, die wieder geöffnet werden, indem sie z.B. erstere über "hinaus fließende" Transitionen darstellt und letztere wie in Abbildung 4.8 mit expliziter Transition und Knoten. Diese feingranulare Differenzierung verschlechtert jedoch die intuitive Verständlichkeit des Sankey-Diagramms. Daher wurde auf die Umsetzung einer gesonderten Darstellung von schließenden Ticket-Status verzichtet.

Änderungen bei Variante 2 des Prototyps

Bereits bei dieser einfachen (wenige verschiedene Pfade, einfacher Trac-Standard-Workflow) Visualisierung eines Ticketflusses tritt ein Problem dieser Darstellung hervor. Sie wird schnell unübersichtlich. Viele Pfade überlagern sich, und es ist schwer zu erkennen, welche Knoten sie miteinander verbinden.

Aufgrund der Korridor-Bildung ausgehend von einem Startknoten und entlang der inneren Knoten entstehen hier keine Überlagerungen. Das Problem tritt erst bei der jeweils letzten Transition jedes Pfades auf, die zu den Endknoten führt. Werden diese Transitionen entfernt, gewinnt die Visualisierung deutlich an Übersichtlichkeit. Abbildung 4.9 zeigt denselben Ticketfluss ohne finale Transitionen wie Abbildung 4.8. Übriggeblieben sind in dieser Darstellung alle Transitionen, die Ticketstatus-Änderungen repräsentieren. An den Startknoten sowie an den inneren Knoten existieren nun im Gegensatz zur Variante 1 des Prototyps Bereiche, die nicht von ausgehenden Transitionen "weiterverarbeitet" werden. Diese Bereiche sind implizit mit dem gleichnamigen Endknoten in der letzten Spalte verbunden.

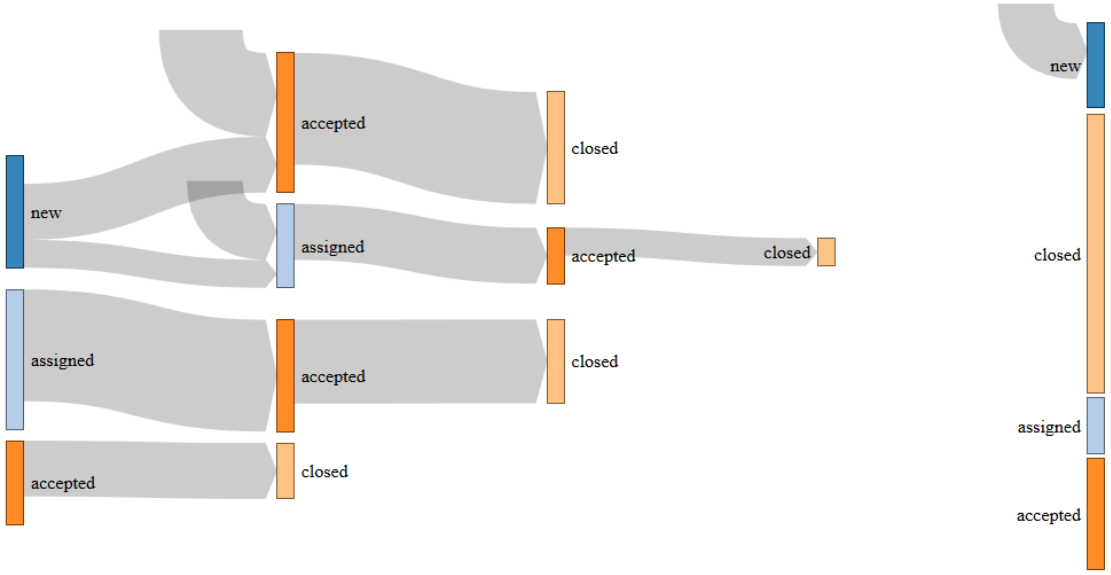

Abbildung 4.9: Variante 2 des zweiten Prototyps

Dem Vorteil der Übersichtlichkeit steht allerdings die Verletzung der Energieerhaltungs-Konvention bei Sankey-Diagrammen gegenüber. Sankey-Diagramme besitzen Quellen (hier: Startknoten) und Senken (hier: Endknoten). Für die dazwischenliegenden Knoten (hier: innere Knoten) gilt: Die Summe der Eingänge ist gleich der Summe der Ausgänge. Die Variante 2 des Prototyps hält diese Regel nicht explizit ein.

Hinweise zur Visualisierung

Wie oben bereits erläutert stellt die erste Spalte anhand der Höhe der Startknoten die Verteilung der Ticket-Status zu Beginn des Betrachtungs-Zeitraums dar. Die letzte Spalte visualisiert analog die Verteilung zum Ende des Zeitraums. Nun liegt die Vermutung nahe, dass die dazwischenliegenden Spalten diesen Zeitraum in gleich lange Intervalle

unterteilen. Das ist aber gerade nicht der Fall. Die Pfade der Sankey-Darstellung erzeugen lediglich eine Halbordnung und stellen die durch den Pfad repräsentierten Tickets zeitlich in eine Vorher-Nachher-Beziehung. Die folgende Liste verdeutlicht zulässige und unzulässige Interpretationen der Visualisierung:

- Ein Pfad sagt nicht aus, wann genau die Status-Änderungen bei den einzelnen Tickets stattgefunden hat. Insbesondere ist der Abstand zwischen den Spalten konstant und steht nicht in Bezug zum Änderungszeitpunkt.
- Die Spaltenposition von Knoten auf verschiedenen Pfade stellt keine zeitliche Relation dar. Insbesondere lässt sich aus einer weiter rechts stehenden Transition auf einem Pfad und einer weiter links angezeigten Transition auf einem anderen Pfad nicht folgern, dass erstere Status-Änderung zeitlich nach letzterer stattgefunden hat.
- Die in einer Transition zusammengefassten Status-Änderungen einer Gruppe von Tickets müssen nicht zeitgleich stattgefunden haben.
- Ein Pfad lässt nur folgende Schlussfolgerung zu: Die durch ihn repräsentierten Tickets haben jeweils zu unbestimmten Zeitpunkten innerhalb des Betrachtungs-Zeitraums die Status der Knotenliste des Pfades (in dieser Reihenfolge) angenommen.

Eine weitere nicht offensichtliche Besonderheit der Visualisierung betrifft geschlossene Tickets, deren Status sich innerhalb des Betrachtungs-Zeitraums nicht ändert. In Change-Request-Systemen werden Tickets in der Regel nach ihrer Bearbeitung geschlossen, aber nicht gelöscht. Daher entsteht nach und nach ein hoher Bestand an geschlossenen Tickets, die nicht weiter bearbeitet werden, während der Anteil aktiver, d.h. nicht geschlossener, Tickets relativ dazu unter der Annahme eines konstanten Ticketdurchsatzes abnimmt. Eine Darstellung aller Tickets des Change-Request-Systems im Sankey-Diagramm würde sehr hohe, wenig aussagekräftige Knoten mit dem Status “closed” in der ersten und letzten Spalte erzeugen, während die interessanten Ticketflüsse der aktiven Tickets auf einer kleinen, dem Verhältnis zwischen aktiven und geschlossenen Tickets entsprechenden Fläche angezeigt würden.

Um dieser mit der Zeit der Nutzung des Change-Request-Systems fortschreitenden Degradierung der Visualisierung entgegenzuwirken, bezieht der zweite Prototyp in beiden Varianten geschlossene Tickets, deren Status sich im Betrachtungs-Zeitraum nicht ändert, nicht in die Darstellung ein.

Interpretation der Visualisierung

Die Interpretation der im Sankey-Diagramm visualisierten Status-Änderungen der Tickets eines Change-Request-Systems überlässt der Prototyp vollständig dem Anwender. Er nutzt dazu seine Kenntnisse über den der Darstellung zugrundeliegenden Softwareentwicklungs-Prozess, um beispielsweise Abweichungen oder ungewöhnliche Vorgänge zu erkennen. Die Bereitstellung einer automatisierten Unterstützung bei der Interpretation ist nicht trivial. Sie erfordert die Entwicklung und Evaluierung eines Modells, das die Zusammenhänge zwischen den Daten eines Change-Request-Systems und

dem tatsächlichen Zustand des widergespiegelten Softwareentwicklungs-Prozesses herstellt und beschreibt. Ein solches Modell wurde im Rahmen dieser Diplomarbeit nicht erstellt, wird aber ggf. von einer nachfolgenden Arbeit behandelt (siehe Abschnitt 7.1). Dennoch werden im Folgenden mögliche Interpretationen der Visualisierung anhand einiger Beispiele beschrieben. Die dabei gefolgerten Schlüsse erscheinen dem Diplomanden plausibel, wurden jedoch nicht systematisch entwickelt oder validiert:

Als Basis für die Beispiele dient folgendes Szenario: Angenommen, eine Organisation verwendet das Scrum-Prozessmodell zur Entwicklung von Software und nutzt ein Change-Request-System zur Verwaltung des Sprint-Backlogs (vgl. Abbildung 4.2). In diesem Change-Request-System sei ein Workflow mit folgenden möglichen Status definiert: “ausstehend”, “in Analyse”, “im Entwurf”, “im Test” und “erledigt”. Dabei sei der direkte Wechsel von jedem Status in jeden anderen erlaubt. Der Scrum-Prozess sieht vor, dass das Scrum-Team mit der Bearbeitung eines Tickets beginnt, wodurch sich sein Status von “ausstehend” zu “in Analyse” ändert. Daraufhin wird ein oder mehrere Male die Abfolge “in Analyse”, “im Entwurf” und “im Test” durchlaufen, bis das Ticket schließlich den Status “erledigt” erhält.

Abweichung von einem vorgegebenen Prozess

In der Visualisierung des Prototyps kann der Anwender “gute” (Prozess-einhaltende) von “schlechten” (Prozess-verletzenden) Pfaden unterscheiden. “Gute” Pfade sind diejenigen, die die im Szenario beschriebene Abfolge von Ticket-Status widerspiegeln. Ebenfalls potenziell “gut” sind Pfade, die ein Suffix bzw. ein Präfix dieser Abfolge abbilden. Hier findet die Bearbeitung der Tickets nur zum Teil innerhalb des in der Visualisierung gewählten Betrachtungs-Zeitraums statt. Ob die Bearbeitung der Tickets außerhalb des Betrachtungs-Zeitraums dem Prozess entspricht, ist unbestimmt. “Schlechte” Pfade sind alle übrigen: Z.B. ein Pfad direkt von “ausstehend” zu “erledigt” oder ein Pfad ohne den Status “im Test”.

Die Schlussfolgerungen, die der Prototyp-Anwender aus dem Vorhandensein von “schlechten” Pfaden ziehen kann, können ohne die oben angesprochene Modellbildung und -evaluierung kaum generisch hergeleitet werden. Weitere, nicht in der Visualisierung sichtbare Faktoren beeinflussen die Schlussfolgerungen ebenso wie die Rolle des Prototyp-Anwenders.

Ein Beispiel: Besitzen die vom Wunschprozess abweichenden “schlechten” Pfade einen kleinen Anteil am Gesamtbild, wird der Prozess weitgehend eingehalten und es sind keine Maßnahmen durch den Projektleiter oder Softwareprozess-Manager erforderlich. Ist der Anteil der “schlechten” Pfade jedoch hoch, sollte der Projektleiter nach den Gründen für die Abweichung vom Prozess forschen. Möglicherweise passt der Prozess nicht optimal zum Projekt. Vielleicht sind viele Tickets derart trivial, dass die Mitarbeiter den Aufwand, den Fortschritt des Tickets im Change-Request-System zu erfassen im Vergleich zum Aufwand zur Bearbeitung des Tickets als unverhältnis hoch empfinden, kurzerhand das Ticket bearbeiten und es direkt vom Status “ausstehend” in den Status “erledigt” versetzen. In diesem Fall könnte der Softwareprozess-Manager den Prozess modifizieren und z.B. den direkten Übergang von “ausstehend” zu “erledigt” für triviale Tickets erlauben.

Dieses Beispiel verdeutlicht den Bedarf, die im Sankey-Diagramm dargestellten “Füll-

stände" und Transfermengen um weitere Informationen (wie in diesem Fall die Aufwandsabschätzung der Tickets) zu ergänzen.

Stockender Prozess

Eine Lagerbildung, d.h. Anhäufung von Tickets, in einem nicht-schließenden Zustand kann auf Probleme im Prozess hindeuten. In einem Scrum-Prozess bearbeitet das gesamte Scrum-Team in der Regel zu einem Zeitpunkt genau ein aktives Ticket. Eine Lagerbildung z.B. im Ticket-Status "in Analyse" weist also auf eine Abweichung vom Prozess hin: Es werden mehrere Tickets gleichzeitig bearbeitet. In einem System, in dem kontinuierlich Tickets erstellt und bearbeitet werden, wie etwa einem Kundenbetreuungssystem, kann eine solche Lagerbildung darauf hinweisen, dass mehr Tickets erstellt als erledigt werden und ggf. nicht genug (etwa personelle) Ressourcen für die Bearbeitung der Tickets zur Verfügung stehen. In einem nicht-iterativen, von Phasen geprägten Softwareentwicklungs-Modell, wie etwa dem klassischen Wasserfallmodell, kann eine Lagerbildung wiederum typisch für den Prozess und ganz normal sein. Z.B. befinden sich etwa alle in Tickets formulierten Anforderungen zu Beginn des Projekts im Status "in Analyse".

Auch hier ist die Diagnose, die aus der Visualisierung des Prozesses gefolgert werden kann, stark Kontext-abhängig.

Gewonnene Erkenntnisse

Die obigen Überlegungen zur Interpretation der Visualisierung zeigen, dass die Ticketfluss-Darstellung durchaus auf Probleme im Softwareentwicklungs-Prozess hinweisen kann. Um die Analyse zu verbessern, sollte die Visualisierung um weitere Kontextinformationen ergänzt werden. Der nächste Schritt, aus der Darstellung des Prozesses Ursachen für Probleme im Prozess und Maßnahmen für deren Lösung abzuleiten, ist jedoch nur schwer automatisierbar und beruht auf dem Expertenwissen des Anwenders.

Die Variante 2 des Prototyps wurde im Rahmen des "GDIS Info Cafe" in einer 20-minütigen Präsentation einigen Führungskräften der Generali Deutschland Informatik Services GmbH vorgestellt. Ziel dieser Veranstaltung war es, einerseits den Kooperationspartner GDIS über Entwicklungen in der Forschungsgruppe Software Construction ins Bild zu setzen. Andererseits bietet sie den Studenten die Möglichkeit, die eher theoretischen Ideen ihrer Abschlussarbeit mit den praktischen Erfahrungen aus der Softwareindustrie abzugleichen. Das in der der Präsentation folgenden Diskussion erhaltene Feedback wird hier wiedergegeben:

Der Ansatz, vorhandene Daten aus Softwareentwicklungs-Werkzeugen und insbesondere aus Change-Request-Sytseme zu visualisieren und zur Prozessanalyse zu verwenden, wurde grundsätzlich positiv bewertet. Es kam allerdings schnell die Frage auf, warum ausgerechnet der Ticketstatus und dessen Veränderung im Zentrum der Visualisierung steht. Wie oben bereits beschrieben, sind die möglichen Schlussfolgerungen aus der Visualisierung uneindeutig.

Stattdessen schlugen die Diskussionsteilnehmer vor, andere Ticket-Aspekte als den Status zu visualisieren, die konkreter interpretiert werden können. Sie verdeutlichten dies anhand von Szenarien:

Kundenbetreuung mit Service-Level-Agreement

Es sei in einem Service-Level-Agreement mit einem Kunden vereinbart worden, dass Fehlermeldungen innerhalb einer Frist bearbeitet werden. Weiterhin werde bei der Meldung des Fehlers anhand der Beschreibung eine Einschätzung vorgenommen, welche Abteilung innerhalb der Organisation für die Behebung des Fehlers zuständig ist. Diese Information wird in einer Eigenschaft des Tickets, einem Feld namens "zuständige Abteilung", festgehalten. Kommt es bei der Erfassung des Fehlers zu einer Fehleinschätzung, welche Abteilung zuständig ist, verbleibt er zunächst in der Abarbeitungswarteschlange der falschen Abteilung, bis er dort bearbeitet werden soll. Nun fällt die initiale Fehleinschätzung auf und der Fehler wird der richtigen Abteilung zur Bearbeitung zugewiesen. Ein solcher Ablauf kann die Bearbeitungszeit signifikant erhöhen und ggf. die Einhaltung des Service-Level-Agreements gefährden.

Abbildung 4.10 zeigt eine Visualisierung der Ticketeigenschaft "zuständige Abteilung". Wie oben beschrieben, ist es "gut", wenn die zuständige Abteilung anhand der Fehlerbeschreibung direkt bei der Fehlererfassung richtig eingeschätzt wird. Dies ist in der Visualisierung durch fehlende Änderungspfade erkennbar. Viele Änderungspfade deuten hingegen auf häufige Fehleinschätzungen hin. Abbildung 4.10 zeigt einen Prozess mit hoher Genauigkeit bei der Einschätzung der für die Fehlerbehebung zuständigen Abteilung.

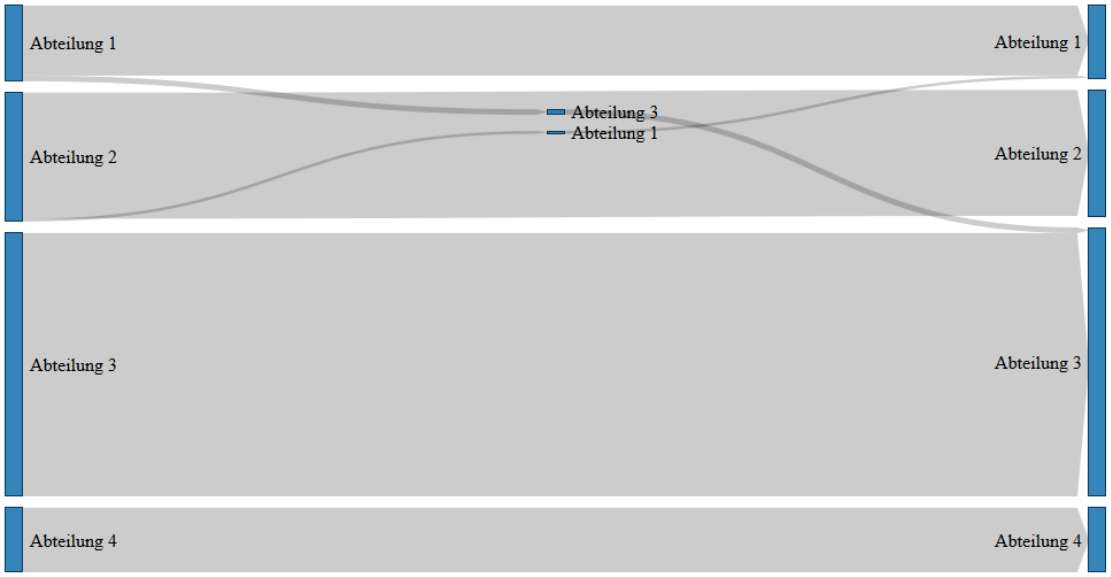

Abbildung 4.10: Visualisierung der Ticketeigenschaft "zuständige Abteilung"

Gegenüberstellung "Fehler gefunden durch" vs "Fehler entstanden in"

In diesem Szenario wird die Verwendung eines V-Modells zur Softwareentwicklung vorausgesetzt. Dabei stehen Phasen und Aktivitäten, in denen Fehler entstehen bzw. gemacht werden können, Prüfungen und Tests gegenüber, die diese Fehler aufdecken sollen. Fehler im Anforderungsdokument sollten etwa durch ein Anforderungsdokument-Review gefunden werden. Modultests finden Fehler in der Codierung. Die Quelle des Fehlers und

die Phase oder Aktivität (Fehleraufdecker), die ihn aufgedeckt hat, wird dabei in den Ticketeigenschaften “Fehler entstanden in” bzw. “Fehler gefunden durch” notiert. Die Visualisierung wurde wie folgt skizziert: Links im Sankey-Diagramm befinden sich mit den Fehlerquellen bezeichnete Knoten. Rechts stehen Knoten mit den Bezeichnungen der Fehleraufdecker. Jeder Fehlerquelle wird ein Fehleraufdecker zugeordnet, indem sie horizontal auf gleicher Höhe dargestellt werden. Die Pfade im Diagramm stellen dar, wie viele Tickets die Fehlerquelle und den Fehleraufdecker besitzen, die der Pfad miteinander verbindet.

Horizontale Pfade sind in dieser Darstellung “gute” Pfade, denn sie verbinden die Phase, die den Fehler enthält mit der Prüfung, die den Zweck hat, ihn aufzudecken. Sie zeigen also, dass der Prozess funktioniert. Geschwungene, s-förmige Pfade sind “schlechte” Pfade: Ein Architekturfehler etwa, der erst im Entwurfs-Review entdeckt wird, und dazu führt, dass zunächst die Architektur korrigiert und anschließend der bereits erstellte Entwurf neu angefertigt werden muss.

4.8 Konsolidierte Anforderungen

In diesem Abschnitt werden die Anforderungen an die Werkzeugunterstützung *River* formuliert. Die ursprünglichen, an den ersten Prototypen gerichteten Anforderungen, gelten weiter. Sie sind hier erneut wiedergegeben:

- (RB1) Um den Einsatz und die Evaluierung des Werkzeugs in existierenden Softwareentwicklungs-Projekten zu erleichtern und eine hohe Akzeptanz zu erreichen, soll das Werkzeug aus Sicht der Software-Entwickler eines Projekts “nichtstörend” sein (vgl. [Joh01]. D.h. es soll von den Entwicklern keinen über ihre normalen Tätigkeiten hinausgehenden zusätzlichen Aufwand erfordern, wie etwa die manuelle Eingabe von Informationen über ihre Aktivitäten im Rahmen des Softwareentwicklungs-Prozesses.
- (Q1) Die Einarbeitungszeit in das Werkzeug soll für mit dem Softwareentwicklungs-Prozess vertraute Personen kurz sein.
- (Q2) Der Konfigurationsaufwand vor und während der Benutzung des Werkzeugs durch die Stakeholder soll gering sein.
- (Q3) Die Visualisierung soll übersichtlich und leicht verständlich sein.
- (FA1) Das Werkzeug soll den Benutzer bei der Analyse des Softwareentwicklungs-Prozesses unterstützen. Dazu zeigt es Informationen über den Prozess-Zustand an und hilft, Abweichungen vom gewünschten Prozess zu erkennen und zu bewerten.
- (FA2) Das Werkzeug soll Daten aus Change-Request-Systemen erheben, weiterverarbeiten und visualisieren.

Darüber hinaus werden die Anforderungen anhand der mit den beiden Prototypen gewonnenen Erkenntnissen wie folgt ergänzt:

- (FA3) Das Werkzeug soll verschiedene Ticketeigenschaften visualisieren können, nicht nur den Ticket-Status.

- (FA4) Das Werkzeug soll die in den Prototypen erprobte Sankey-Diagramm-Darstellung um weitere Kontextinformationen ergänzen, die die Analyse des Prozesses unterstützen.

Schließlich folgen noch einige, nicht aus den Prototypen abgeleitete Anforderungen, die die Benutzbarkeit und Wiederverwendbarkeit des Werkzeugs steigern sollen:

- (FA5) Das Werkzeug soll über eine Schnittstelle verfügen, über die Change-Request-Systeme das Werkzeug über Änderungen an ihren Tickets informieren können, um die im Werkzeug erhobenen Daten zu aktualisieren.
- (Q4) Das Werkzeug soll mit verschiedenen Change-Request-Systemen zusammenarbeiten können.
- (RB2) Das Werkzeug soll aus wiederverwendbaren Komponenten bestehen und in der Technologie JavaEE entwickelt werden. Insbesondere sollen die Datenerhebung, die Aufbereitung der Daten und die Visualisierung durch einzelne Komponenten bereitgestellt werden, um eine ggf. später erfolgende Integration des Werkzeugs in MeDIC zu erleichtern.

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.

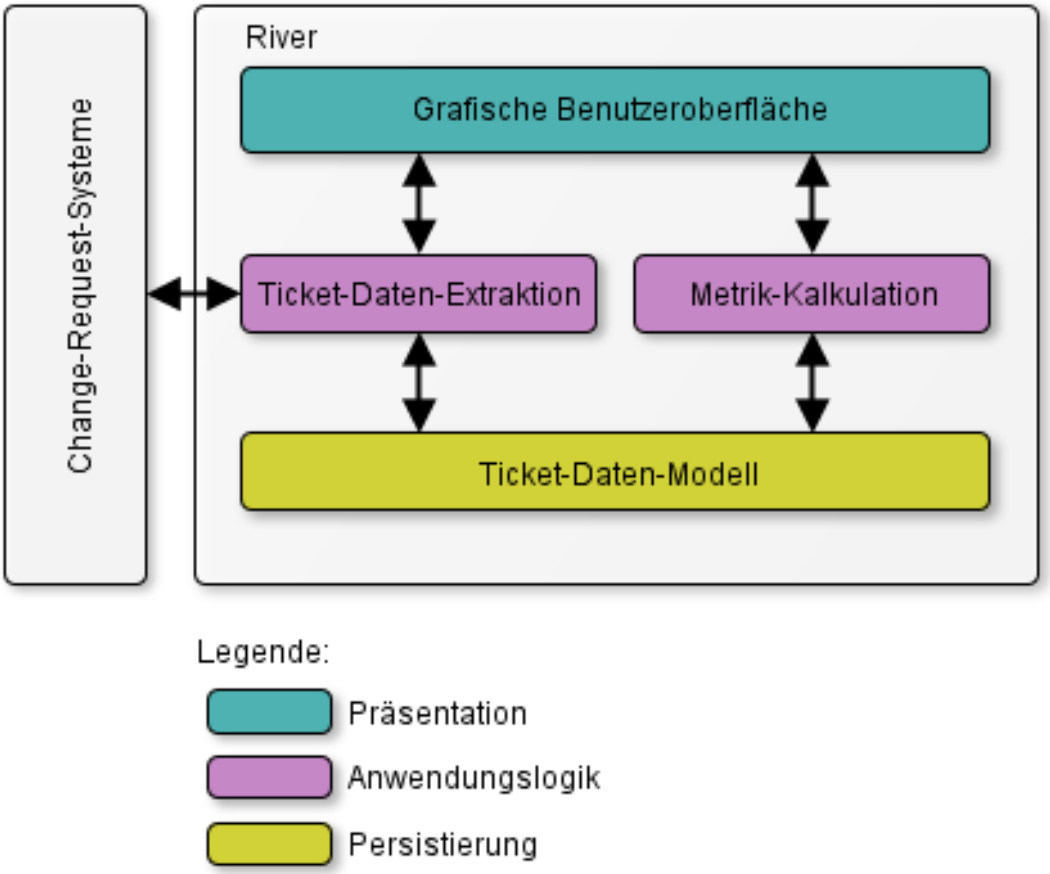

Abbildung 5.1: *River*: Technologieunabhängige Schichten-Architektur

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

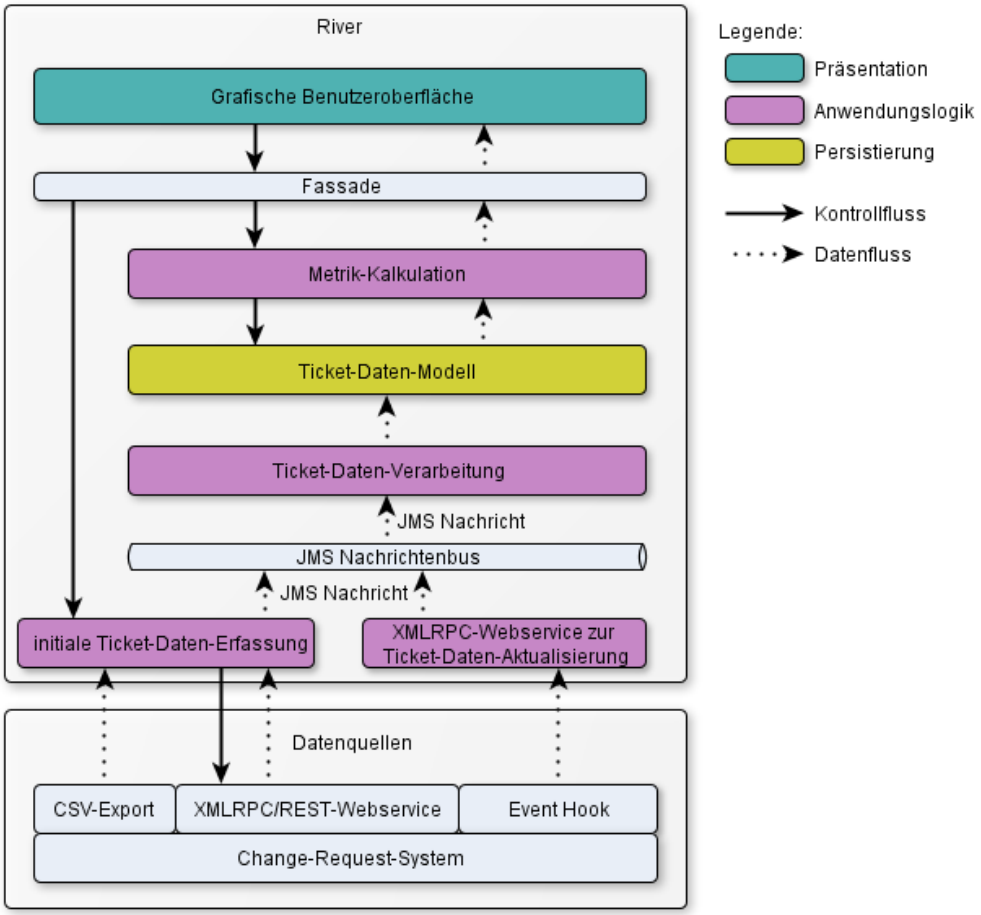

Abbildung 5.2: *River*: Technologieabhängige Architektur

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).

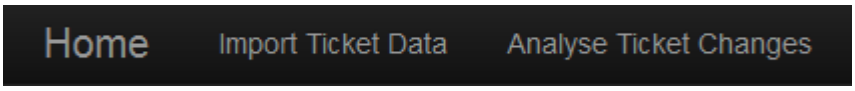

Abbildung 5.3: Navigationsleiste in *River*

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.

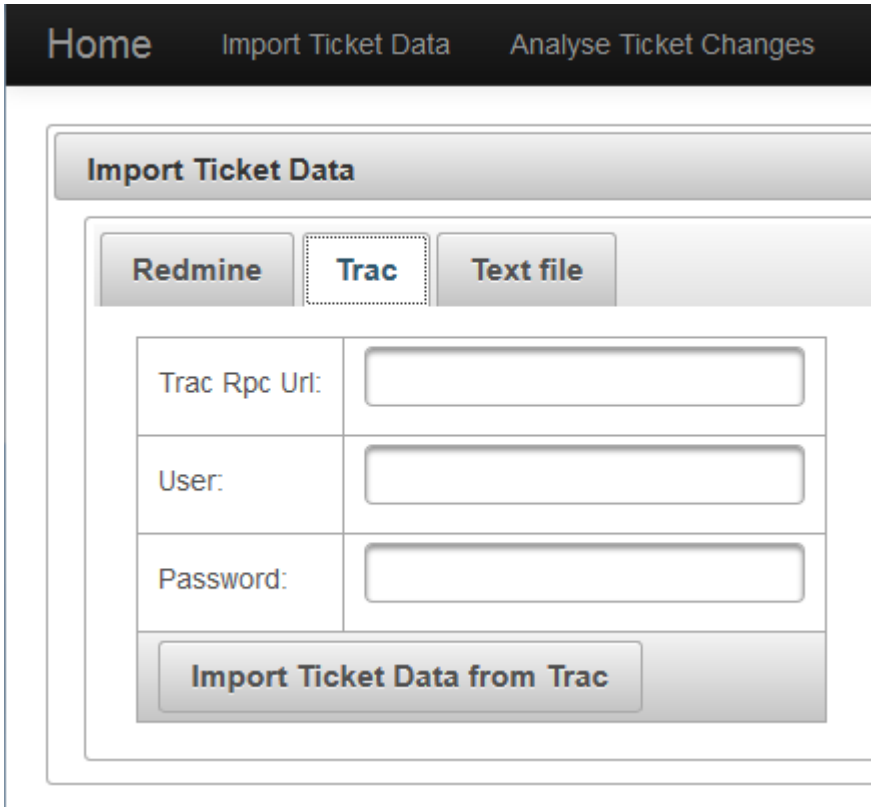

Abbildung 5.4: Import aus einem Trac-System

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".

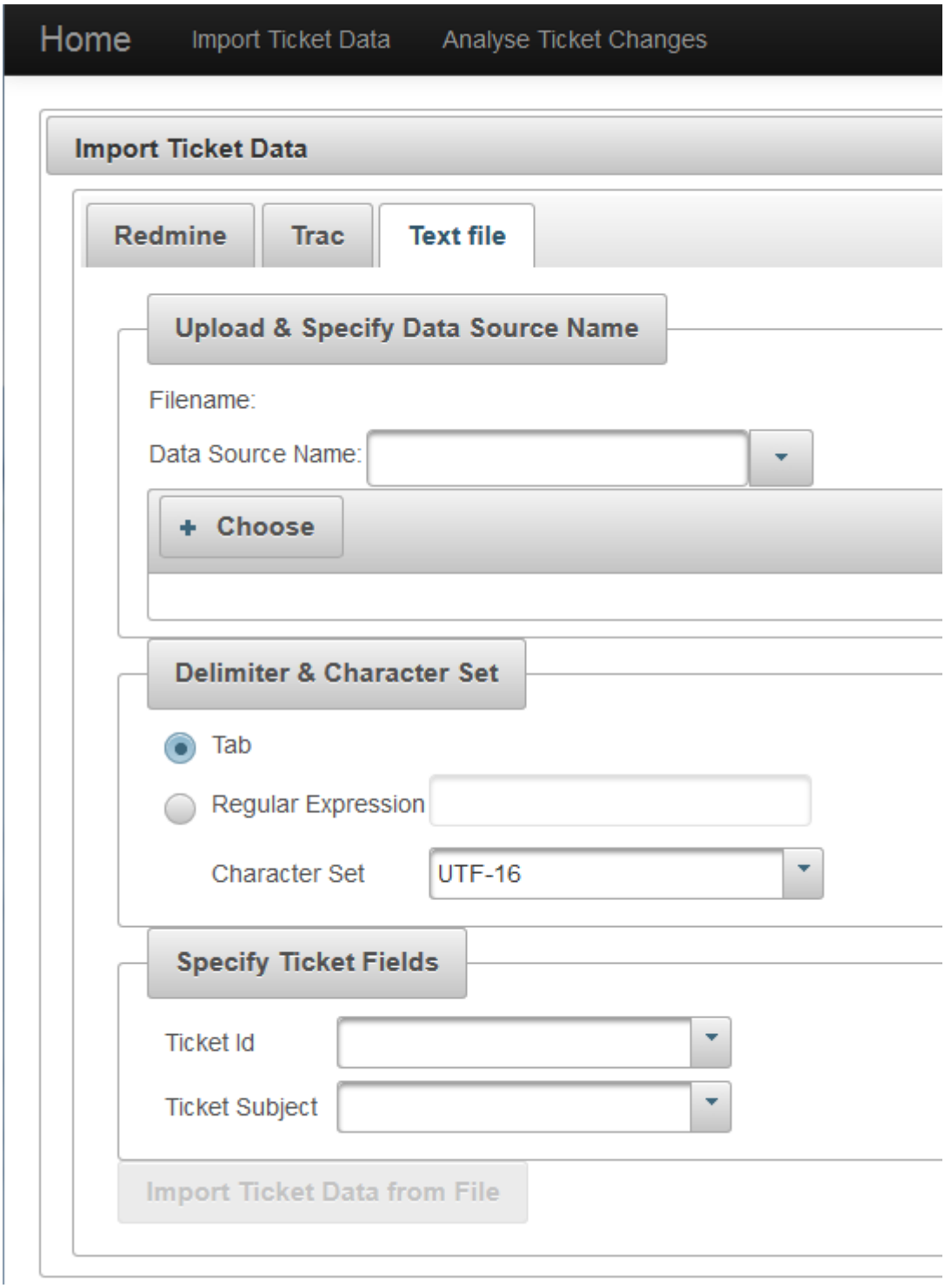

Abbildung 5.5: Import aus einer exportierten Ticket-Daten-Datei

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.

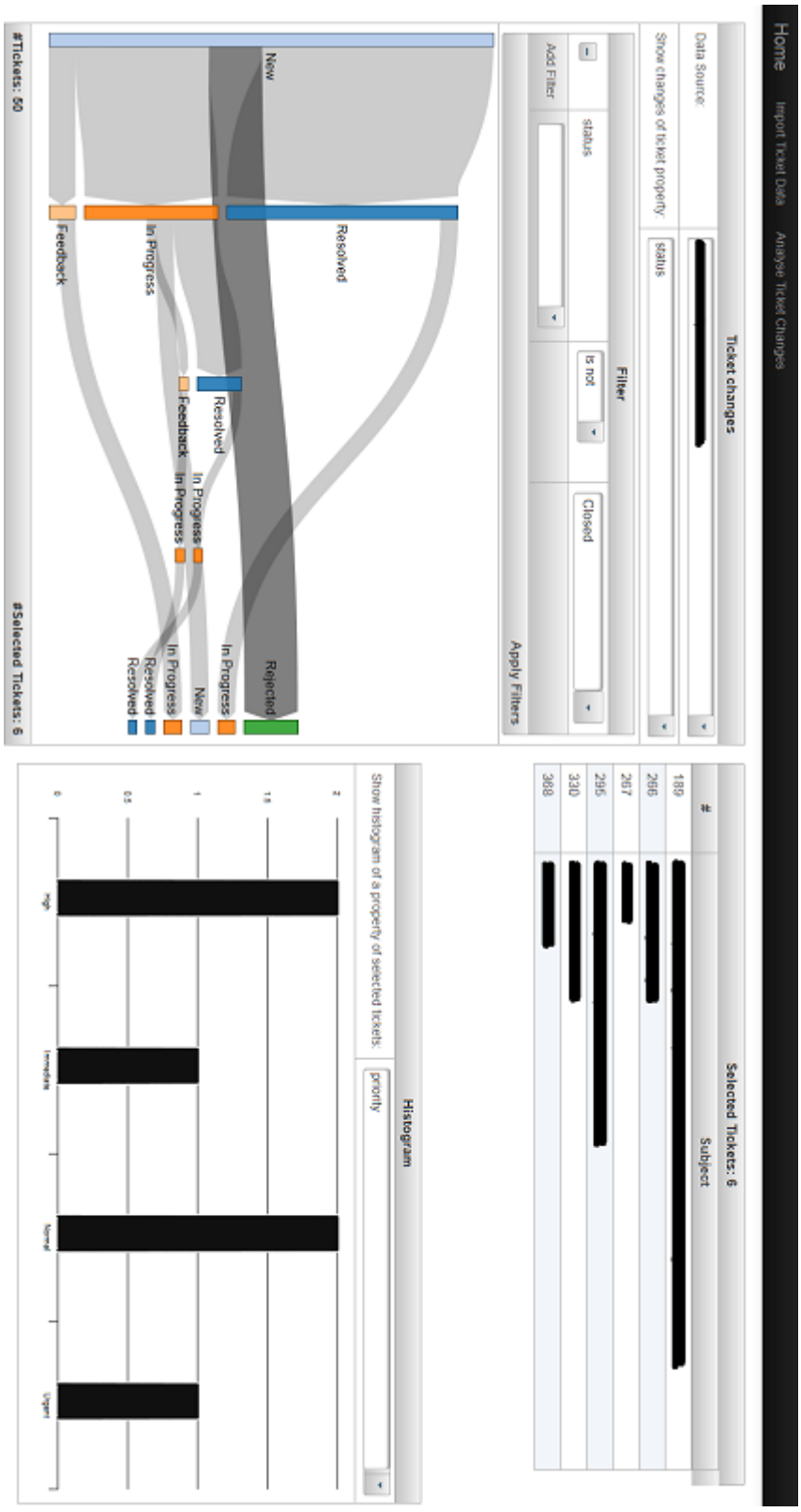

Abbildung 5.6: Analyse mit *River*

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.

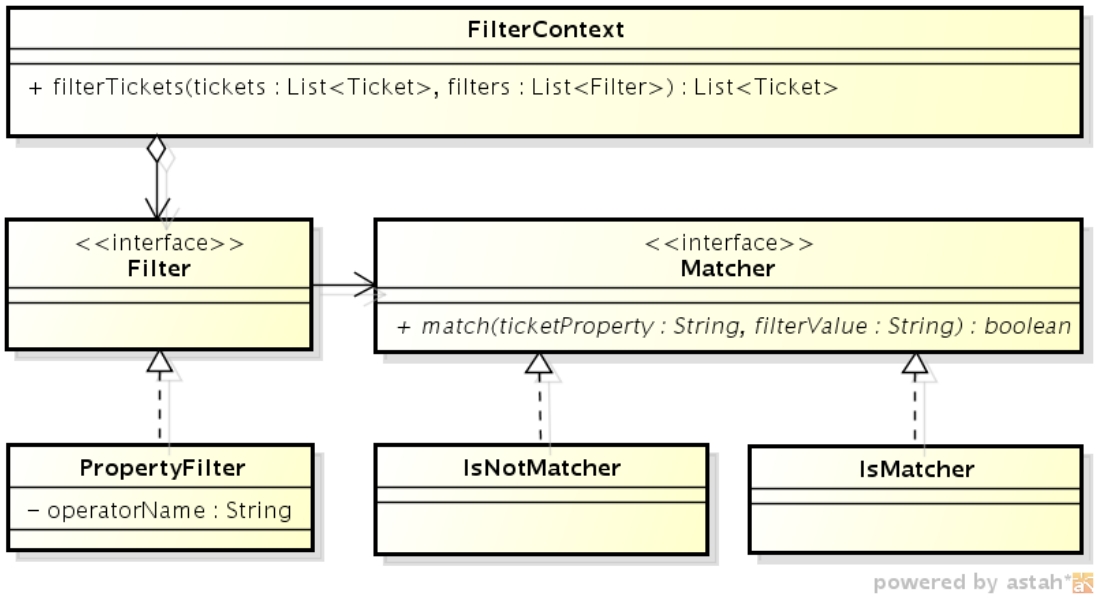

Abbildung 5.7: Klassendiagramm der Filterung

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

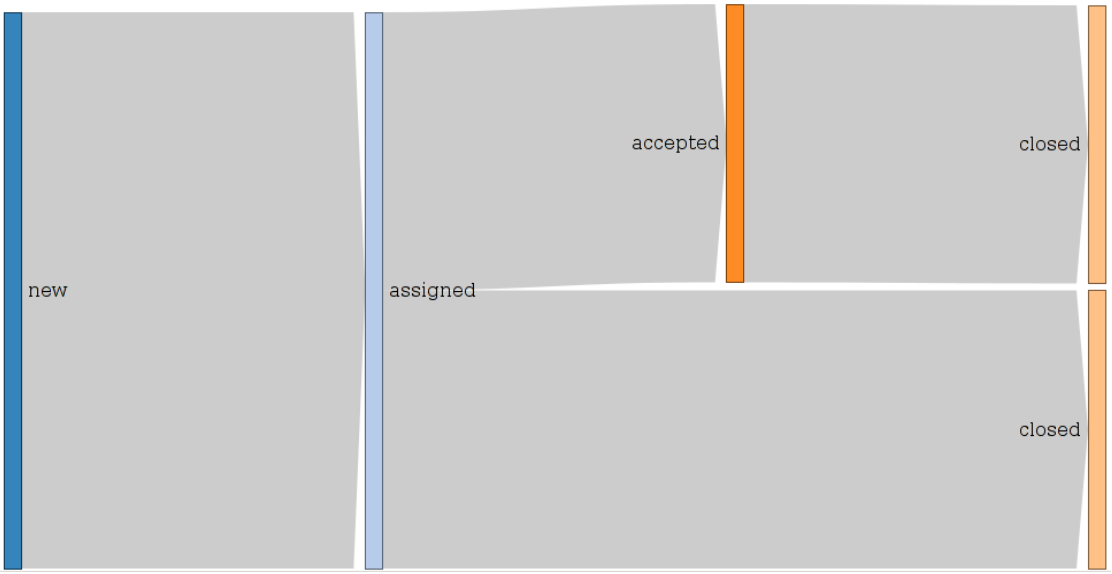

Abbildung 5.8: Visualisierung der Ticket-Änderungen im Sankey-Diagramm

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.

6 Evaluation

Inhaltsangabe

Dieses Kapitel beschreibt die Ziele, die Vorgehensweise und die Ergebnisse der Evaluation zu der in dieser Diplomarbeit entstandenen Werkzeugunterstützung *River*.

6.1 Ziele und Vorgehensweise

Ziel der Evaluation ist die Beantwortung der Frage, ob und inwiefern das Werkzeug *River* die Stakeholder bei der Analyse und Optimierung eines Softwareentwicklungs-Prozesses unterstützt. Kann ein Stakeholder über die Visualisierung der automatisiert erhobenen Ticket-Daten und Ticket-Historien eines Change-Request-Systems Rückschlüsse auf Vorgänge im Softwareentwicklungs-Prozess, der laut der zentralen Hypothese dieser Arbeit durch das Change-Request-System abgebildet wird, ziehen? Falls ja, soll die Evaluation insbesondere ermitteln, welche Rückschlüsse möglich sind?

Die Evaluation wurde in Form von qualitativen, semi-strukturierten Einzelinterviews im Rahmen von zwei Sitzungen mit jeweils einem Mitarbeiter der beiden Kooperationspartner dieser Diplomarbeit durchgeführt. Die befragten Mitarbeiter decken die identifizierten Rollen der Stakeholder, Projektleiter und Softwareprozess-Manager, ab (vgl. Abschnitt 4.2). Der Aufgabenbereich von Stakeholder A beinhaltet beide Rollen. Als Geschäftsführer eines mittelständischen IT-Unternehmens führt er Projekte durch und besitzt gleichzeitig ein hohes Interesse an der Optimierung der im Unternehmen eingesetzten Prozesse. Stakeholder B erfüllt die Rolle eines Softwareprozess-Managers in einem großen IT-Unternehmen und ist mit vielen dort durchgeführten Softwareentwicklungs-Projekten vertraut. Beide Stakeholder sind Experten für die im Rahmen der Evaluation mit *River* zu untersuchenden Softwareentwicklungs-Prozesse aus ihrem jeweiligen Unternehmen.

Das qualitative Interview wurde aus zwei Gründen einer quantitativen Evaluation über einen standardisierten Fragebogen vorgezogen:

1. Der Fokus der Evaluation liegt darauf herauszufinden, welche Rückschlüsse auf einen Softwareentwicklungs-Prozess mittels *River* möglich sind. Das qualitative Interview mit einem Experten eignet sich insbesondere dazu, explorativ Erkenntnisse zu gewinnen, die dem Interviewer unbekannt sind (siehe [May08]). Hier ermöglicht es mit Hilfe des befragten Experten, vom Diplomanden vermutete Zusammenhänge zwischen der Visualisierung und den dem Experten vertrauten

Softwareentwicklungs-Prozess zu bestätigen oder zu widerlegen, sowie unerwartete Zusammenhänge zu erkennen, die der Experte erkennt.

2. Die Stärke des standardisierten Fragebogens, die erhobenen Daten statistisch aufzubereiten, kommt durch die geringe Teilnehmer-Anzahl (zwei) an der Evaluation nicht zum Tragen.

Ablauf des Interviews

Vor dem Interview wurden die durch den jeweiligen Kooperationspartner zur Verfügung gestellten Daten des entsprechenden Change-Request-Systems in das Werkzeug importiert und Vermutungen über Eigenschaften des Softwareentwicklungs-Prozesses auf Basis der Visualisierungen angestellt.

Zu Beginn des Interviews wurde der Experte mit der Bedienung des Werkzeugs und der Semantik der Visualisierung vertraut gemacht. Im zweiten Schritt wurden dem Experten mit Hilfe des Werkzeugs die Visualisierungen präsentiert, zu denen zuvor Vermutungen über ihre Aussagekraft bezüglich des Softwareentwicklungs-Prozesses angestellt wurden. Der Experte sollte diese Vermutungen - auf Basis seiner genauen Kenntnisse des dargestellten Prozesses - kommentieren und entweder bestätigen oder widerlegen. Schließlich sollte der Experte das Werkzeug selbständig und frei verwenden, und den Prozess weiter analysieren. Hierbei sollten Erkenntnisse über weitere Zusammenhänge zwischen Visualisierung und Prozess gewonnen werden.

Während des Interviews wurden zusätzlich die Kommentare des Experten zum Werkzeug selbst festgehalten. Darunter fallen Bemerkungen über die Benutzbarkeit, Verständlichkeit und Eignung des Werkzeugs, der Visualisierungen oder einzelner Anzeigeelemente, sowie Hinweise auf mögliche Verbesserungen.

Die nächsten beiden Abschnitte fassen die Ergebnisse der Interviews mit den beiden Stakeholdern zusammen.

6.2 Evaluation mit Kooperationspartner A

Die Analyse mit Kooperationspartner A wurde anhand der Ticket-Daten eines Redmine-Change-Request-Systems zu einem Software-Projekt durchgeführt. Die etwa 300 Tickets wurden über den *Redmine-Adapter* in das Werkzeug importiert. Die folgenden Szenarien zeigen Probleme im Softwareentwicklungs-Prozess auf, die mit Hilfe von *River* identifiziert wurden.

Szenario: Status-Fluss

Die Visualisierung des Ticket-Status-Flusses (Abbildung 6.1) zeigte keinen hauptsächlich verfolgten Workflow-Pfad. Mit Hilfe der auf die Eigenschaft “assigned to” eingestellten Histogramm-Detailansicht war erkennbar, dass die Status-Pfade stark mit dem Ticket-Bearbeiter korrelieren. Dies weist auf eine nicht standardisierte Nutzung des Change-Request-Systems hin, was der Interviewte bestätigte.

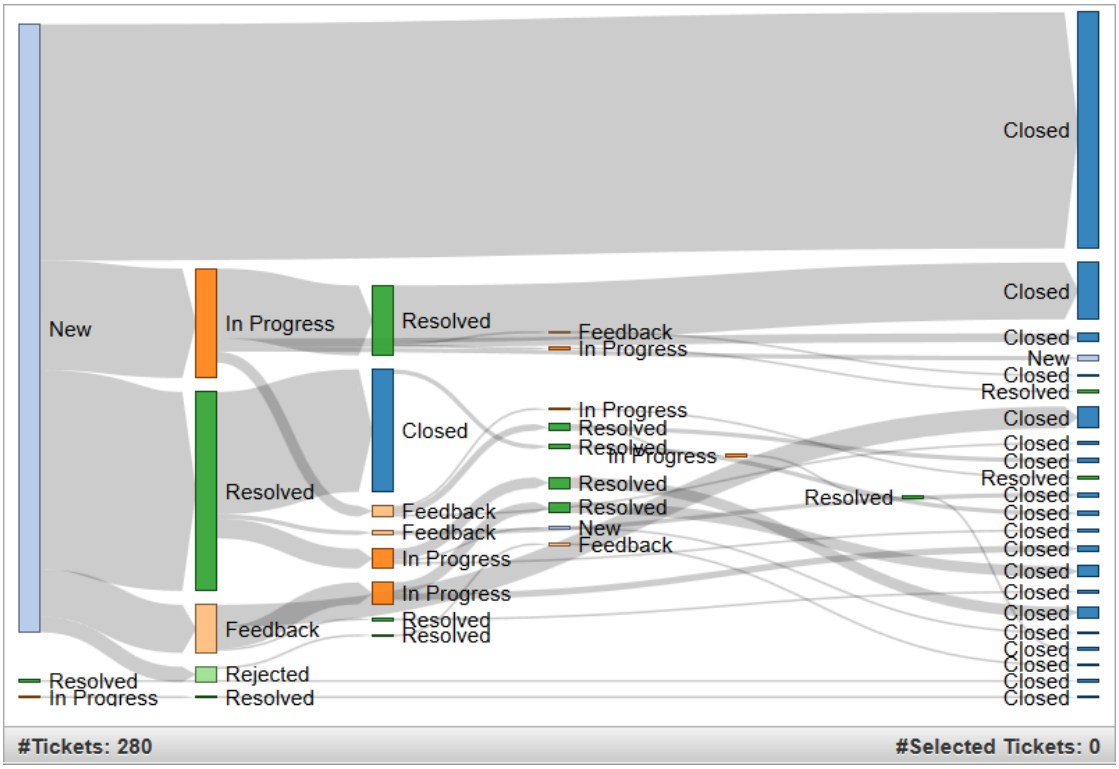

Abbildung 6.1: Evaluations-Szenario: Status-Fluss

Szenario: Ticket-Bearbeitungs-Fortschritt

In dieser Ansicht (Abbildung 6.2) ist die Nutzung der prozentual angegebenen Ticketeigenschaft “Ticket-Bearbeitungs-Fortschritt” (“done ratio”) visualisiert. Der Wertebereich zwischen 0% und 50% wird fast nicht genutzt, während der Wert 90% gehäuft auftritt. Die Eigenschaft scheint dazu genutzt zu werden, “fast erledigte” Tickets zu markieren, wird aber nicht zur Kennzeichnung von Tickets genutzt, deren Bearbeitung sich in einem frühen Stadium befindet. Möglicherweise ist der Bearbeitungs-Fortschritt zu einem frühen Zeitpunkt schwerer zu beurteilen als zu einem späten. Dieses Muster bei der Nutzung der Ticketeigenschaft “Ticket-Bearbeitungs-Fortschritt” war dem Interviewten zuvor nicht bewusst.

Szenario: Geschätzter Aufwand

In nur wenigen der etwa 300 Tickets wurde die Ticketeigenschaft “geschätzter Aufwand” verwendet (Abbildung 6.3). Dies weist auf eine nicht auf den Prozess zugeschnittene Ticketdefinition (z.B. eine nicht oder nur teilweise modifizierte Standard-Ticketdefinition) im Change-Request-System hin. In der Regel ist es jedoch sinnvoll, die Konfiguration des Change-Request-Systems an den Prozess anzupassen und nicht umgekehrt. Der Interviewte erläuterte, dass sich Qualitäts-Kontroll-Maßnahmen zum Projekt-Controlling, die auf geschätzten Aufwänden basieren, im Unternehmen derzeit im Aufbau befinden und schrittweise in den Projekten eingeführt werden.

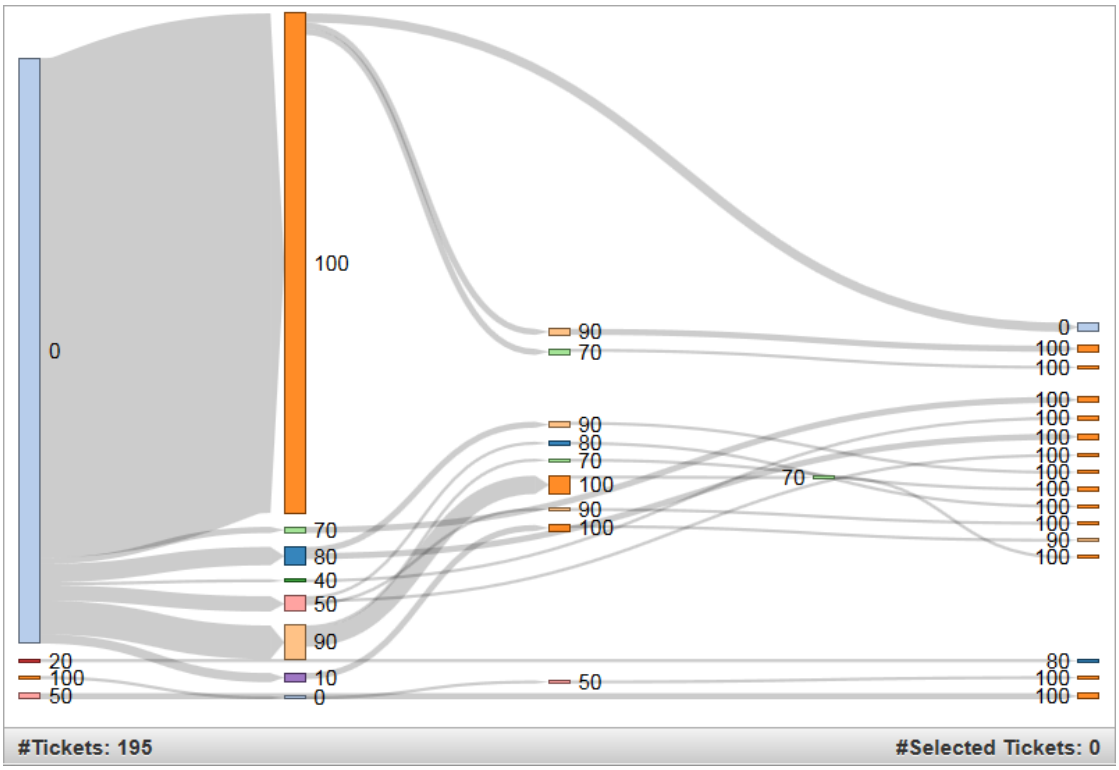

Abbildung 6.2: Evaluations-Szenario: Ticket-Bearbeitungs-Fortschritt

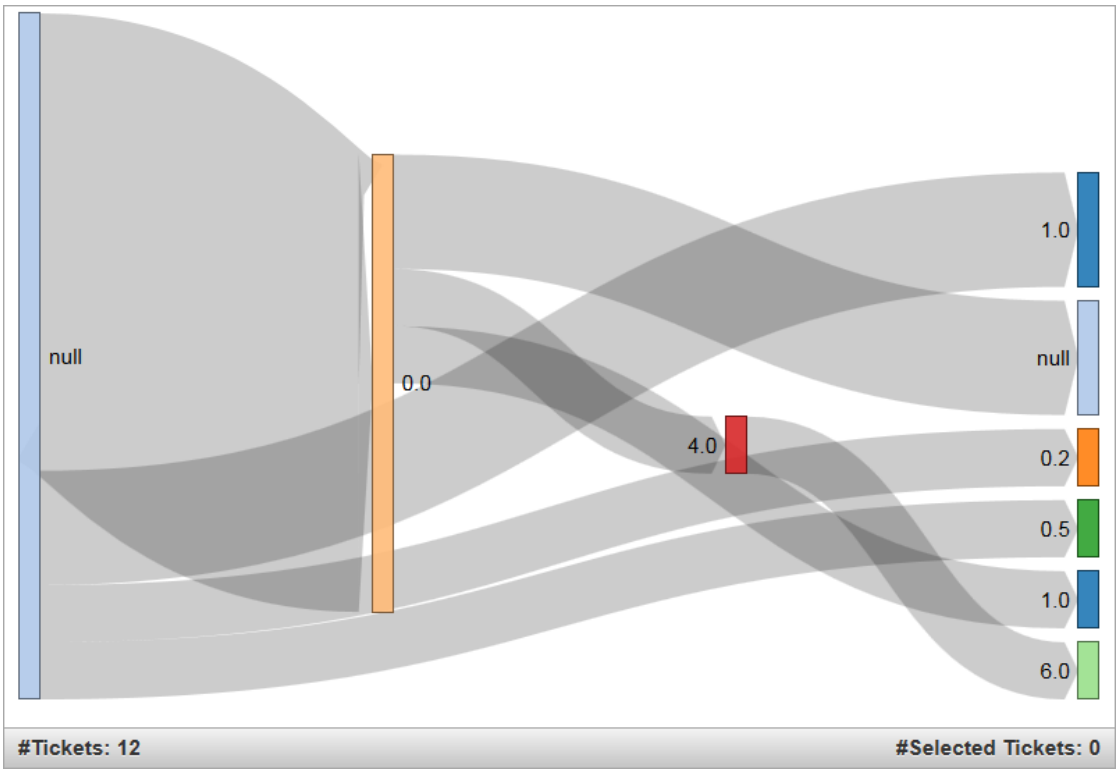

Abbildung 6.3: Evaluations-Szenario: Geschätzter Aufwand

Bewertung des Werkzeugs

- Nach einer kurzen Erläuterung der Semantik der Sankey-Diagramme zur Darstellung von Ticketeigenschafts-Flüssen war der Interviewte schnell in der Lage die Visualisierung zu verstehen und zu interpretieren.
- Aufgrund seiner Rolle im Unternehmen war der Interviewte mit den Tickets vertraut. Mehrfach fielen dem Interviewten nach der Auswahl der zu untersuchenden Ticketeigenschaft und der Definition eines Filters zunächst unerwartete Pfade im Sankey-Diagramm auf. Die zusätzliche Information der in der Detailansicht angezeigten Ticket-Themen half ihm bei der Bewertung dieses Pfades.
- Der Interviewte vermisste die konsolidierte Darstellung der aktuellen Ticketzustände im Sankey-Diagramm, wie sie im zweiten Prototypen verwendet wurde. Die Unterteilung der Tickets trotz identischem finalen Wert in der betrachteten Ticketeigenschaft sei nicht intuitiv verständlich.
- Es sollte die Möglichkeit geben, über Filterregeln den Betrachtungs-Zeitraum der zu analysierenden Tickets einzuschränken.
- Das Change-Request-System des Kooperationspartners A wird von (externen) Kunden verwendet um Fehler zu berichten, die dann von Mitarbeitern behoben werden. Daneben erstellen aber auch die Mitarbeiter selbst im Rahmen interner Prozesse Tickets. Der Interviewte schlug einen Filter vor, der auf der Gruppenzugehörigkeit (extern/intern) der Ticket-Ersteller basiert, um etwa ausschließlich die von Kunden erstellten Tickets zu analysieren.
- Die Oberfläche des Change-Request-System des Interviewten ist lokalisiert. Eine Ticketeigenschaft heißt z.B. “Geschätzter Aufwand”. Diese Lokalisierung ist in den importierten Daten nicht widergespiegelt. Daher findet der Benutzer die gleiche Ticketeigenschaft in *River* unter dem Begriff “estimated_hours” wieder. *River* sollte ebenfalls lokalisierte Ticketeigenschaftsnamen verwenden.
- Neu erstellte Tickets, die bezüglich der betrachteten Ticketeigenschaft nicht geändert werden, sollten ebenfalls dargestellt werden.

6.3 Evaluation mit Kooperationspartner B

Die folgenden Szenarien beschreiben die Erkenntnisse über die Softwareentwicklungs-Prozesse bei Kooperationspartner B, die durch die Analyse der Ticket-Daten und Ticket-Historien des dort genutzten Change-Request-Systems ClearQuest gewonnen wurden. Die Analysen basieren auf monatlichen Abzügen des Ticket-Status. Diese Abzüge wurden über die Datei-Import-Funktion in *River* importiert. Das Change-Request-System verwaltete etwa 30000 Tickets.

Szenario: Einschätzung der Zuständigkeit

Bei der Erfassung von Produktionsfehlern wird das zugehörige Ticket einem Teilprojekt zugeordnet, das für die Bearbeitung zuständig ist. Ist diese Zuordnung zunächst fehler-

haft, verzögert sich die Bearbeitung des Tickets ggf. erheblich und kann etwa die Einhaltung von Bearbeitungsfristen, die mit einem Kunden in einem Service Level Agreement getroffen wurden, gefährden. In der Visualisierung ist erkennbar, dass diese Zuordnung in aller Regel korrekt vorgenommen wird. Bei lediglich 26 von insgesamt ca. 30000 Tickets musste sie korrigiert werden (Abbildung 6.4). Es sind keine Hinweise auf eine Abweichung vom gewünschten Prozess erkennbar.

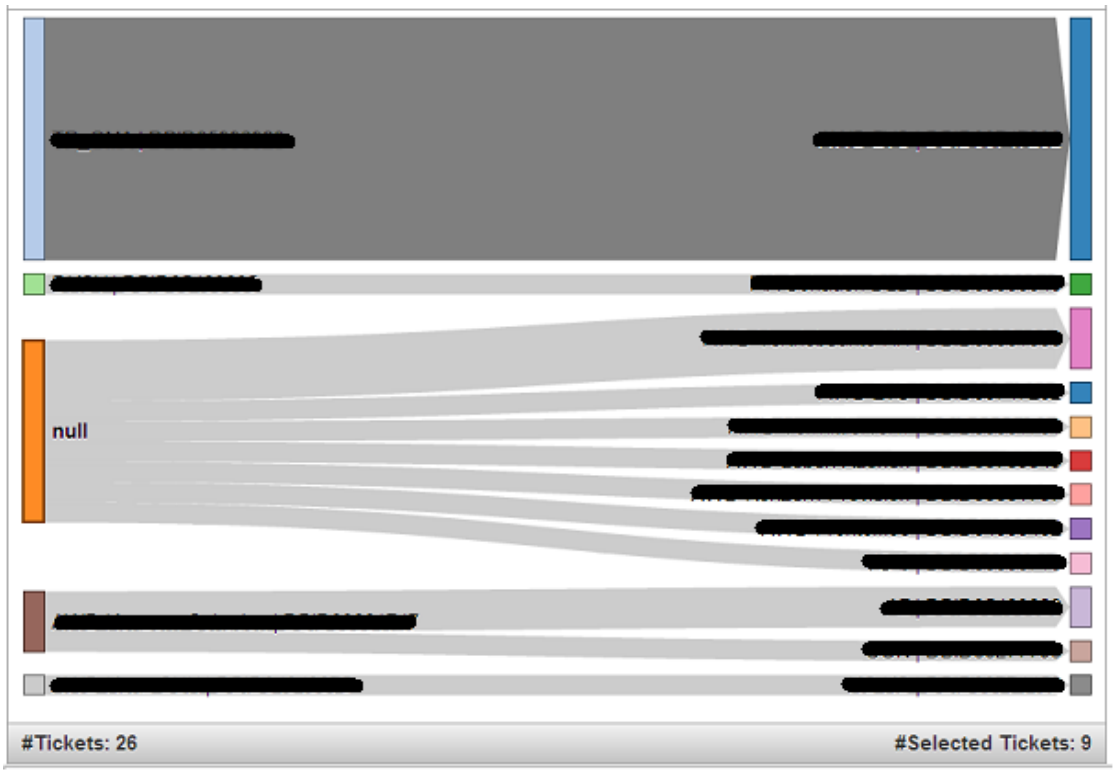

Abbildung 6.4: Evaluations-Szenario: Einschätzung der Zuständigkeit

Szenario: Häufigste Fehlerursache

In diesem Szenario zeigt die Visualisierung (Abbildung 6.5), dass viele Fehler ihren Ursprung in den Bereichen Code und Daten haben, den meisten Fehlern aber keine Ursache zugeordnet wird oder werden kann. Möglicherweise sind die verfügbaren Kategorien, die in diesem Feld eingetragen werden können, ungeeignet. Der Stakeholder sollte die Ursache für die ungewöhnliche Verwendung dieser Ticketeigenschaft weiter untersuchen.

Bewertung des Werkzeugs

- Analog zum Interview mit dem Experten des Kooperationspartners A war der Interviewte nach einer kurzen Einführung in die Bedienung des Werkzeugs und die Semantik der Sankey-Diagramm-Darstellung in der Lage, die Visualisierung des Prozesses zu verstehen und zu interpretieren.
- Er bewertete die Visualisierung nach der Einführung als intuitiv verständlich.

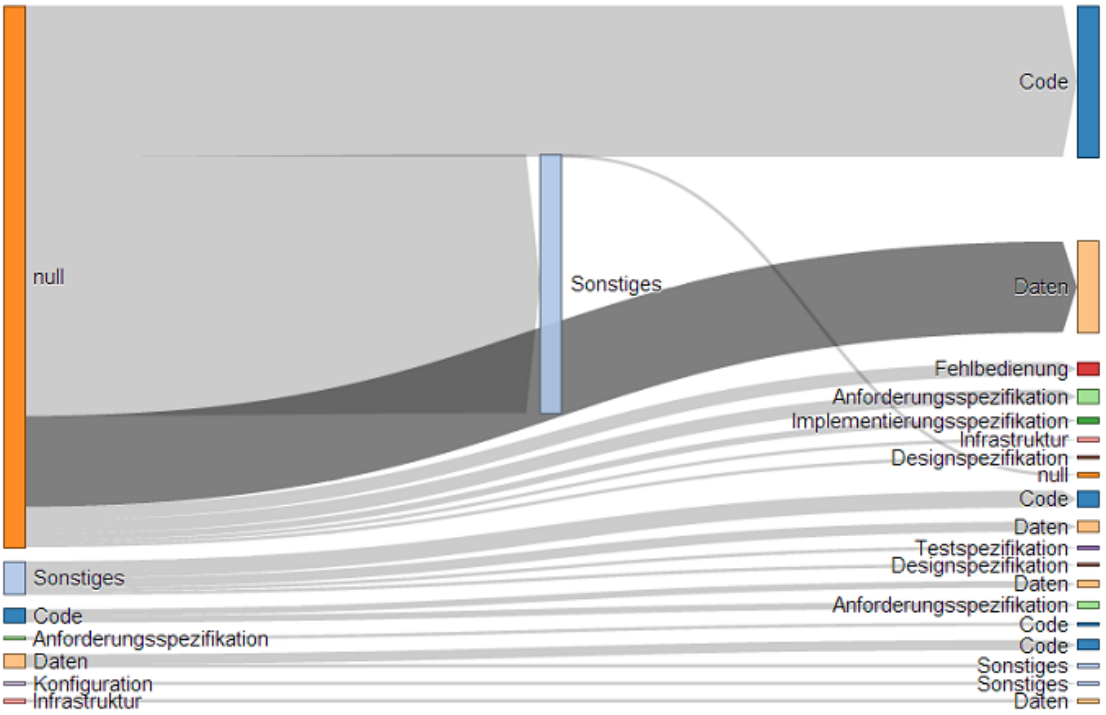

Abbildung 6.5: Evaluations-Szenario: Häufigste Fehlerursache

- Ebenso wie der Experte des Kooperationspartners A hatte der hier Interviewte den Wunsch nach einer konsolidierten Darstellung der Verteilung der Tickets am Ende des Betrachtungs-Zeitraums.

- Oft sei eine Abstraktion von den konkreten Ticketeigenschaftswerten erwünscht. Die möglichen Werte der Ticketeigenschaft "Status" lassen sich etwa in die beiden Kategorien "Offenes Ticket" und "Geschlossenes Ticket" einteilen. Das Werkzeug sollte Filter auf Basis dieser Einteilung erlauben und in der Visualisierung Tickets derselben Kategorie gemeinsam darstellen können.

6.4 Zusammenfassung

Anhand der gezeigten Szenarien waren die Evaluationsteilnehmer mit Hilfe des Werkzeugs in der Lage, bereits bekannte Prozessabweichungen (vgl. Szenario: kein Standardisierter Prozess) nachzuvollziehen. Ebenso gelang es, die erfolgreiche Umsetzung eines Prozessaspekts nachzuweisen (vgl. Szenario: Einschätzung der Zuständigkeit). Schließlich wurden auch neue Erkenntnisse über die Nutzung eines Change-Request-Systems gewonnen (vgl. Szenario: Ticket-Bearbeitungs-Fortschritt).

Die Werkzeug-Bewertung der beiden Evaluationsteilnehmer weist starke Übereinstimmung auf. Beide weisen darauf hin, dass die symmetrische Darstellung der Verteilung der Tickets gemäß der analysierten Ticketeigenschaft zu Beginn und zum Ende des Betrachtungs-Zeitraums wie sie im Prototyp verwendet wurde der im Werkzeug geänderten Sankey-Diagramm-Darstellung überlegen ist, insbesondere bezüglich der intuitiven

Verständlichkeit. Die Darstellung sollte daher in einer späteren Version von *River* entsprechend angepasst werden.

Die Forderung nach einem Filter für Ticket-Status-Kategorien bzw. Benutzer-Rollen entspricht verallgemeinert der Forderung, Metainformationen über die Semantik von Ticketeigenschaften in den Filtern des Werkzeugs zu verwenden. Diese Metainformationen stammen nicht mehr aus den Tickets selbst, sondern aus sonstigen im Change-Request-System vorhandenen Daten, wie etwa einer Workflow-Definition, die die Information enthält ob ein Ticket-Status in die Kategorie "offen" bzw. "geschlossen" fällt.

Liegen diese Metainformationen erst einmal dem Werkzeug vor, können sie, neben ihrer Verwendung in Filtern, auch zur Abstraktion der Visualisierung genutzt werden. Eine Darstellung des Ticket-Status, die lediglich die Kategorien "offen" und "geschlossen" differenziert, würde etwa den Anteil der Tickets, die in ihrem Lebenszyklus nach einer Schließung wieder eröffnet werden, sehr deutlich von den übrigen Tickets abheben.

Insgesamt gaben beide Evaluationsteilnehmer positive Bewertungen zu dem Werkzeug und seiner Visualisierung der Softwareentwicklungs-Prozesse durch Ticket-Eigenschafts-Änderungs-Flüsse ab. Die Visualisierung sei nach kurzer Erläuterung intuitiv verständlich und die Informationen in den Detailansichten (Ticket-Liste und Histogramm zu einer weiteren Ticketeigenschaft) seien hilfreich bei der Untersuchung einzelner Ticketeigenschafts-Pfade.

7 Zusammenfassung und Ausblick

Inhaltsangabe

In dieser Diplomarbeit wurde die Frage untersucht wie auf Basis grundsätzlich vorhandener, wenn auch ggf. nicht sichtbarer Daten aus Softwareentwicklungs-Werkzeugen die Analyse und Optimierung von Softwareentwicklungs-Prozessen unterstützt werden kann. Dazu wurde die Hypothese aufgestellt, dass zwischen diesen Daten und den Prozessen eine Korrelation existiert. Change-Request-Systeme wurden dabei als primäre Datenquelle identifiziert, da die Durchführung vieler Prozess mit Hilfe solcher Systeme geführt und überwacht wird.

Ausgehend von bestehenden Visualisierungs-Methoden zur Darstellung von Prozessen in industriellen Betrieben wurde anhand von Prototypen untersucht, inwiefern sich diese Visualisierungen sinnvoll von industriellen auf Softwareentwicklungs-Prozesse übertragen lassen. Ein erster Prototyp visualisierte den Prozess anhand des Workflows des Change-Request-Systems. Ähnlich wie in industriellen Prozessleitständen ist der Prozess wiedererkennbar dargestellt. Die Darstellung erlaubt lediglich die Verfolgung von Änderungen des Ticket-Status, und lässt sich nicht einfach um weitergehende Informationen erweitern.

Anhand des zweiten Prototyps wurde eine alternative Visualisierung der Ticket-Status-Veränderungen mittels Sankey-Diagrammen erprobt. Diese Darstellung erwies sich genau wie die Visualisierung des Workflows als intuitiv verständlich. Trotz seiner einfachen Struktur (spaltenweise Anordnung der Knoten, lediglich horizontal verlaufende Kanten) lassen sich in diesem Diagrammtyp ebenso wenig wie im Workflow zusätzliche komplexe Kontext-Informationen (wie etwa Liniendiagramme zur Darstellung eines Trends) an den Graphelementen anbringen, ohne dass die Übersichtlichkeit und Verständlichkeit der Darstellung stark reduziert wird.

Eine Evaluation des zweiten Prototyps bei den Kooperationspartnern dieser Diplomarbeit ergab, dass der bis jetzt im Fokus stehende Ticket-Status nicht das einzige interessante, in Change-Request-Systemen vorhandene Datum sei, aus dem Rückschlüsse auf den Prozess gezogen werden können.

Im Anschluss an die Prototypen wurde daher das Werkzeug *River* entwickelt. Ein Datenmodell für Daten aus Change-Request-Systemen bildet die Basis für Metriken, die die Änderungen von Ticketeigenschaften beschreiben. Diese Metriken werden auf der Benutzeroberfläche des Werkzeugs visualisiert und ermöglichen eine Analyse des den Daten zugrunde liegenden Softwareentwicklungs-Prozesses. Das Datenmodell wird über eine Daten-Import- und Daten-Aktualisierungs-Schnittstelle mit Ticket-Daten aus Change-Request-Systemen befüllt. Zur Anbindung an ein Change-Request-System dient ein system-spezifischer Datenquellen-Adapter.

Die die Diplomarbeit abschließende Evaluation des Werkzeugs zeigte, dass die visualisierten Ticket-Daten tatsächlich Rückschlüsse auf die zugrundeliegenden Prozesse erlauben. Damit wird die eingangs aufgestellte Hypothese über eine Korrelation zwischen Ticket-Daten und Prozessen bestätigt.

7.1 Ausblick

Im Rahmen dieser Diplomarbeit wurde herausgefunden, dass die Umsetzung von Softwareentwicklungs-Prozessen mit den Ticket-Änderungs-Daten aus einem im Prozess eingesetzten Change-Request-Systems korrelieren. Das wirft die Frage nach der Beschaffenheit dieser Korrelation auf. Die Interpretation der visualisierten Ticket-Änderungen wurde vollständig dem Anwender des Werkzeugs überlassen. Dieser setzt sein Wissen über z.B. den gewünschten Ablauf eines Prozesses ein, um anhand der visualisierten Ticket-Änderungs-Daten auf Prozess-Abweichungen zu schließen. Dieser Ansatz weist zwei offensichtliche Probleme auf:

- Es besteht die Möglichkeit, dass der Anwender mit Hilfe des Werkzeugs lediglich bereits auf anderen Wegen erkannte Prozess-Probleme ermittelt. Dies geschieht leicht, wenn er sich von seinem Wissen über unerwünschte Abläufe im Prozess bei der Analyse leiten lässt und so das Werkzeug nur zur Bestätigung bereits bekannter Probleme nutzt. Bekannte Probleme sind (oder sollten) zu diesem Zeitpunkt bereits Gegenstand gegensteuernder Maßnahmen sein. Es sind also gerade die noch nicht erkannten Prozess-Probleme auf die ein hilfreiches Werkzeug zur Unterstützung der Prozessanalyse hinweisen sollte.

- Ein unerfahrener Benutzer, etwa ein Projektleiter der gerade sein erstes Projekt durchführt, ist vermutlich zu weniger Schlussfolgerungen aus den Visualisierungen des Werkzeugs in der Lage als ein erfahrener Projektleiter, der gut mit dem Softwareentwicklungs-Prozess und seiner gewünschten Umsetzung vertraut ist. Gerade der unerfahrene Anwender würde aber von Hinweisen des Werkzeugs auf Missstände im Prozess besonders profitieren.

Die offene Frage ist daher: Wie kann dieser Interpretationsvorgang des Werkzeug-Anwenders, z.B. mit Hilfe weiterer Metriken automatisiert oder unterstützt werden? Dieser Frage geht die kommende, englischsprachige Master-Arbeit “Visually Assisted Mining for Smells in Change Request Systems” von E. Gjino nach[Gji13].

Weiter wurden in dieser Arbeit lediglich Daten aus Change-Request-Systemen verwendet. In Softwareentwicklungs-Prozessen werden aber zahlreiche weitere Standard-Werkzeuge verwendet: Versionskontrollsysteme wie Git oder Subversion, Continuous-Integration-Systeme wie Hudson oder Bamboo, Werkzeuge zur Messung der Code-Qualität wie Sonar. Welche Informationen über den Softwareentwicklungs-Prozess können aus den Daten dieser Systeme gewonnen werden? Und welche Erkenntnisse ergibt die Kombination der Daten aus verschiedenen Werkzeugen?

A Anhang

A.1 Technische Realisierung des ersten Prototyps

Der erste Prototyp besteht aus einem in der Programmiersprache Python geschriebenen Plugin für das Change-Request-System Trac sowie einem Java-Backend, das für die Zeichnung des Graphen verantwortlich ist. Das Trac-Plugin basiert in weiten Teilen auf dem "Hello World 2"-Plugin[1] aus dem Plugin-Entwicklungs-Tutorial-Bereich der Internet-Seite "track-hacks.org"[2]. Die Methoden *get_active_navigation_item* und *get_navigation_items* sorgen für den Eintrag eines Links zur Plugin-Seite in der Hauptnavigation des Trac-Systems. Die Methode *match_request* legt fest, unter welcher URL innerhalb des Trac-Systems die Visualisierung des Prototyps abrufbar ist. Abschließend definiert *process_request* das Genshi-Template[3], das zur Darstellung verwendet wird. In diesem Fall handelt es sich um eine einfache HTML-Seite, die ein Bild von einer lokalen URL einbindet.

Hinter dieser URL verbirgt sich ein mit Hilfe der *Java API für XML-Webservices (JAX-WS)* realisierter XMLRPC-Dienst, der eine Grafik des Trac-Standard-Workflow zeichnet und in Form eines PNG-Bilds als HTTP-Antwort zurückliefert (siehe Quelltext A.1). Zur Zeichnung des Workflows wurde das *Java Universal Network/Graph Framework (JUNG)*[4] verwendet. Es erlaubt u.A. die einfache Definition eines gerichteten Graphen in Java, beschriftet Knoten und Kanten und zeichnet den Graph gemäß eines selbst definierbaren oder aus einer vorhandenen Sammlung auswählbaren Layout-Algorithmus.

```
1  package de.rwth.swc.ccharles.crsdb.xmlrpc;
2
3  import java.awt.BasicStroke;
4  import java.awt.Color;
5  import java.awt.Dimension;
6  import java.awt.Image;
7  import java.awt.Point;
8  import java.awt.image.BufferedImage;
9  import java.io.ByteArrayOutputStream;
10 import java.io.IOException;
11
12 import javax.imageio.ImageIO;
13
14 import org.apache.commons.codec.binary.Base64;
15 import org.apache.commons.collections15.Transformer;
```

[1] https://trac-hacks.org/wiki/EggCookingTutorialTrac0.11 abgerufen am 30.05.2013
[2] http://trac-hacks.org/ stellt Subversion-Hosting für von Trac-Nutzern erstellte Trac-Plugins zur Verfügung; abgerufen am 30.05.2013
[3] http://genshi.edgewall.org/ abgerufen am 30.05.2013
[4] http://jung.sourceforge.net/ abgerufen am 31.05.2013

```
16 import org.apache.commons.collections15.functors.ConstantTransformer;
17
18 import edu.uci.ics.jung.algorithms.layout.FRLayout;
19 import edu.uci.ics.jung.graph.Graph;
20 import edu.uci.ics.jung.graph.SparseMultigraph;
21 import edu.uci.ics.jung.graph.util.EdgeType;
22 import edu.uci.ics.jung.visualization.VisualizationImageServer;
23 import edu.uci.ics.jung.visualization.decorators.
       AbstractEdgeShapeTransformer;
24 import edu.uci.ics.jung.visualization.decorators.
       ConstantDirectionalEdgeValueTransformer;
25 import edu.uci.ics.jung.visualization.decorators.EdgeShape;
26 import edu.uci.ics.jung.visualization.renderers.
       GradientVertexRenderer;
27 import edu.uci.ics.jung.visualization.renderers.
       VertexLabelAsShapeRenderer;
28
29 public class WorkflowRenderer {
30
31     public String workflowPicture() {
32
33         // states
34         final State NEW = new State("new");
35         final State CLOSED = new State("closed");
36         final State ASSIGNED = new State("assigned");
37         final State REOPENED = new State("reopened");
38         final State ACCEPTED = new State("accepted");
39
40         Graph<State, Transition> graph;
41
42         VisualizationImageServer<State, Transition> vv;
43
44         // create a simple graph for the demo
45         graph = new SparseMultigraph<State, Transition>();
46
47         graph.addVertex(CLOSED);
48         graph.addVertex(NEW);
49         graph.addVertex(ACCEPTED);
50         graph.addVertex(ASSIGNED);
51         graph.addVertex(REOPENED);
52
53         graph.addEdge(new Transition("reopen"), CLOSED, REOPENED,
54                 EdgeType.DIRECTED);
55
56         graph.addEdge(new Transition("resolve"), NEW, CLOSED,
               EdgeType.DIRECTED);
57         graph.addEdge(new Transition("resolve"), ACCEPTED, CLOSED,
58                 EdgeType.DIRECTED);
59         graph.addEdge(new Transition("resolve"), REOPENED, CLOSED,
60                 EdgeType.DIRECTED);
61         graph.addEdge(new Transition("resolve"), ASSIGNED, CLOSED,
62                 EdgeType.DIRECTED);
```

```
63
64          graph.addEdge(new Transition("accept"), NEW, ACCEPTED,
65                  EdgeType.DIRECTED);
66          graph.addEdge(new Transition("accept"), ACCEPTED, ACCEPTED,
67                  EdgeType.DIRECTED);
68          graph.addEdge(new Transition("accept"), ASSIGNED, ACCEPTED,
69                  EdgeType.DIRECTED);
70          graph.addEdge(new Transition("accept"), REOPENED, ACCEPTED,
71                  EdgeType.DIRECTED);
72
73          graph.addEdge(new Transition("reassign"), REOPENED, ASSIGNED,
74                  EdgeType.DIRECTED);
75          graph.addEdge(new Transition("reassign"), ASSIGNED, ASSIGNED,
76                  EdgeType.DIRECTED);
77          graph.addEdge(new Transition("reassign"), NEW, ASSIGNED,
78                  EdgeType.DIRECTED);
79          graph.addEdge(new Transition("reassign"), ACCEPTED, ASSIGNED,
80                  EdgeType.DIRECTED);
81
82          FRLayout<State, Transition> layout = new FRLayout<State,
               Transition>(
83                  graph);
84          layout.setRepulsionMultiplier(1.5);
85          vv = new VisualizationImageServer<State, Transition>(layout,
               new Dimension(
86                  600, 400));
87          vv.setBackground(Color.white);
88
89          // this class will provide both label drawing and vertex
               shapes
90          VertexLabelAsShapeRenderer<State, Transition> vlasr = new
               VertexLabelAsShapeRenderer<State, Transition>(
91                  vv.getRenderContext());
92
93          Transformer<State, String> state2String = new Transformer<
               State, String>() {
94              public String transform(State v) {
95                  return v.toString();
96              }
97          };
98          vv.getRenderContext().setVertexShapeTransformer(vlasr);
99          vv.getRenderContext().setVertexLabelTransformer(state2String)
               ;
100
101         vv.getRenderer().setVertexRenderer(
102                 new GradientVertexRenderer<State, Transition>(Color.
                    MAGENTA,
103                         Color.WHITE, true));
104         vv.getRenderer().setVertexLabelRenderer(vlasr);
105
106         Transformer<Transition, String> transition2String = new
               Transformer<Transition, String>() {
```

```
107             public String transform(Transition e) {
108                 return e.toString();
109             }
110         };
111         vv.getRenderContext().setEdgeLabelTransformer(
                transition2String);
112         AbstractEdgeShapeTransformer<State, Transition> aesf = new
                EdgeShape.QuadCurve<State, Transition>();
113         aesf.setControlOffsetIncrement(30);
114         vv.getRenderContext().setEdgeShapeTransformer(aesf);
115         vv.getRenderContext().setEdgeStrokeTransformer(
116                 new ConstantTransformer(new BasicStroke(2.5f)));
117
118         ConstantDirectionalEdgeValueTransformer<State, Transition> mv
                = new ConstantDirectionalEdgeValueTransformer<State,
                Transition>(
119                 .5, .5);
120         vv.getRenderContext().setEdgeLabelClosenessTransformer(mv);
121
122
123         Image image = vv.getImage(new Point(300,200), new Dimension
                (600,400));
124
125         BufferedImage bufferedImage = new BufferedImage(600, 400,
126                 BufferedImage.TYPE_INT_RGB);
127
128         bufferedImage.getGraphics().drawImage(image, 0, 0, null);
129
130
131         ByteArrayOutputStream baos = new ByteArrayOutputStream();
132         try {
133             ImageIO.write(bufferedImage, "png", baos);
134         } catch (IOException e1) {
135             e1.printStackTrace();
136         }
137         String base64EncodedImage = Base64.encodeBase64String(baos
138                 .toByteArray());
139
140         System.out.println(base64EncodedImage.length());
141
142         return base64EncodedImage;
143
144     }
145 }
```

Quelltext A.1: Java XMLRPC-Dienst des ersten Prototyps

A.2 Technische Realisierung des zweiten Prototyps

Der zweite Prototyp ist wie bereits der erste Prototyp als Trac-Plugin realisiert. Das Genshi-Template bindet das Data-Driven-Documents-Sankey-Plugin (vgl.Abschnitt 2.4) zur Visualisierung des Sankey-Diagramms ein. Die Berechnung des Modells des Ticket-Status-Fluss-Graphen wird an eine Java Anwendung delegiert, die einen JAX-Webservice zum Abruf der JSON-Darstellung des Graphen implementiert. Die Java Anwendung greift über die Java SQL API und einen SqLite-JDBC-Treiber direkt auf die im lokalen Dateisystem vorhandene SqLite-Datenbank des Trac-Systems zu und extrahiert die Eigenschaft "Status" jedes Tickets und dessen Historie. Diese werden weiter zu einem Modell des Ticket-Status-Fluss-Graphen verarbeitet, das schließlich in die vom Sankey-Plugin verwendete JSON-Darstellung transformiert wird.

A.3 Java Implementierung der Sankey-Metrik

```
1  package de.rwth.swc.gplcrs.controller;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.HashSet;
6  import java.util.LinkedList;
7  import java.util.List;
8  import java.util.Map;
9  import java.util.Map.Entry;
10 import java.util.Set;
11
12 import javax.ejb.EJB;
13 import javax.ejb.Stateless;
14
15 import org.jgrapht.graph.DefaultDirectedWeightedGraph;
16 import org.json.JSONArray;
17 import org.json.JSONException;
18 import org.json.JSONObject;
19
20 import de.rwth.swc.gplcrs.dao.TicketChangeDaoLocal;
21 import de.rwth.swc.gplcrs.dao.TicketDaoLocal;
22 import de.rwth.swc.gplcrs.entity.Ticket;
23 import de.rwth.swc.gplcrs.entity.TicketChange;
24 import de.rwth.swc.gplcrs.sankey.graph.SankeyGraphEdge;
25 import de.rwth.swc.gplcrs.sankey.graph.SankeyGraphNode;
26 import de.rwth.swc.gplcrs.filter.Filter;
27 import de.rwth.swc.gplcrs.filter.FilterContext;
28
29 @Stateless
30 public class SankeyCalculatorBean implements SankeyCalculatorLocal {
31
32     @EJB
33     private TicketChangeDaoLocal ticketChangeDao;
34
35     @EJB
```

```
36      private TicketDaoLocal ticketDao;
37
38      private DefaultDirectedWeightedGraph<SankeyGraphNode,
            SankeyGraphEdge> ticketChangeGraph;
39
40      private List<Ticket> getFilteredTickets(String
            dataSourceIdentifier,
41              List<Filter> filters) {
42
43          List<Ticket> tickets = ticketDao.getAllTickets(
                dataSourceIdentifier);
44          List<Ticket> filteredTickets = FilterContext.filterTickets(
                tickets,
45                  filters);
46          return filteredTickets;
47      }
48
49      private List<TicketChange> getFilteredTicketChanges(
50              String dataSourceIdentifier, String propertyName,
51              List<Filter> filters) {
52
53          List<Ticket> filteredTickets = getFilteredTickets(
                dataSourceIdentifier,
54                  filters);
55
56          Set<Long> filteredTicketIds = new HashSet<Long>();
57
58          for (Ticket filteredTicket : filteredTickets) {
59              filteredTicketIds.add(filteredTicket.getTicketId());
60          }
61
62          List<TicketChange> ticketChanges = ticketChangeDao
63                  .getTicketChangesForProperty(dataSourceIdentifier,
                        propertyName);
64
65          List<TicketChange> filteredTicketChanges = new ArrayList<
                TicketChange>();
66
67          for (TicketChange ticketChange : ticketChanges) {
68              if (filteredTicketIds.contains(ticketChange.getTicketId()
                    )) {
69                  filteredTicketChanges.add(ticketChange);
70              }
71          }
72
73          return filteredTicketChanges;
74      }
75
76      @SuppressWarnings("unchecked")
77      private void calculateChangeGraph(String dataSourceIdentifier,
78              String propertyName, List<Filter> filters) {
79
```

```
80          List<TicketChange> filteredTicketChanges =
                getFilteredTicketChanges(
81                  dataSourceIdentifier, propertyName, filters);
82
83          // build ticket change map. Maps ticket ids to lists of
                String
84          // representing the state changes
85          Map<Long, LinkedList<String>> ticketChangeMap = new HashMap<
                Long, LinkedList<String>>();
86
87          for (TicketChange ticketChange : filteredTicketChanges) {
88              // for each ticket change: if we already have a flow for
                    this
89              // ticket, append the new state
90              if (ticketChangeMap.containsKey(ticketChange.getTicketId
                    ())) {
91                  List<String> ticketFlow = ticketChangeMap.get(
                        ticketChange
92                          .getTicketId());
93                  ticketFlow.add(ticketChange.getNewValue());
94                  // otherwise create a new flow for this ticket id,
                        and start
95                  // the chain with old value -> new value
96              } else {
97                  LinkedList<String> ticketFlow = new LinkedList<String
                        >();
98                  ticketFlow.add(ticketChange.getOldValue());
99                  ticketFlow.add(ticketChange.getNewValue());
100                 ticketChangeMap.put(ticketChange.getTicketId(),
                        ticketFlow);
101             }
102         }
103
104         // build the graph state Map. States are represented by their
                ticket
105         // change history
106
107         ticketChangeGraph = new DefaultDirectedWeightedGraph<
                SankeyGraphNode, SankeyGraphEdge>(
108                 SankeyGraphEdge.class);
109         Map<List<String>, SankeyGraphNode> nodeMap = new HashMap<List
                <String>, SankeyGraphNode>();
110
111         for (Entry<Long, LinkedList<String>> ticketChangeEntry :
                ticketChangeMap
112                 .entrySet()) {
113             LinkedList<String> clonedTicketChange = (LinkedList<
                    String>) ticketChangeEntry
114                     .getValue().clone();
115             while (!clonedTicketChange.isEmpty()) {
116                 // create node in ticketflow graph if it does not
                        exist already
```

```
117                 SankeyGraphNode sankeyGraphNode = new SankeyGraphNode
                        (
118                         clonedTicketChange);
119                 if (!ticketChangeGraph.containsVertex(sankeyGraphNode
                        )) {
120                     ticketChangeGraph.addVertex(sankeyGraphNode);
121                     // put the node into the node map so it can be
                            retrieved by
122                     // later ticketchanges regarding the same node
123                     nodeMap.put(clonedTicketChange, sankeyGraphNode);
124                 } else {
125                     sankeyGraphNode = nodeMap.get(clonedTicketChange)
                            ;
126                 }
127                 // add ticket id to the set of tickets represented by
                        this
128                 // node
129                 sankeyGraphNode.getTicketIds().add(ticketChangeEntry.
                        getKey());
130
131                 clonedTicketChange = (LinkedList<String>)
                        clonedTicketChange
132                         .clone();
133                 clonedTicketChange.removeLast();
134             }
135         }
136
137         for (Entry<Long, LinkedList<String>> ticketChangeEntry :
                ticketChangeMap
138                 .entrySet()) {
139             LinkedList<String> clonedTicketChange = (LinkedList<
                    String>) ticketChangeEntry
140                     .getValue().clone();
141             while (clonedTicketChange.size() >= 2) {
142                 LinkedList<String> targetState = (LinkedList<String>)
                        clonedTicketChange
143                         .clone();
144                 SankeyGraphNode targetNode = nodeMap.get(targetState)
                        ;
145                 clonedTicketChange = (LinkedList<String>)
                        clonedTicketChange
146                         .clone();
147                 clonedTicketChange.removeLast();
148                 LinkedList<String> sourceState = (LinkedList<String>)
                        clonedTicketChange
149                         .clone();
150                 SankeyGraphNode sourceNode = nodeMap.get(sourceState)
                        ;
151
152                 SankeyGraphEdge edge;
153                 if (ticketChangeGraph.containsEdge(sourceNode,
                        targetNode)) {
```

```
154                     edge = ticketChangeGraph.getEdge(sourceNode,
                            targetNode);
155                     double weight = ticketChangeGraph.getEdgeWeight(
                            edge);
156                     ticketChangeGraph.setEdgeWeight(edge, weight + 1)
                            ;
157                 } else {
158                     edge = ticketChangeGraph.addEdge(sourceNode,
                            targetNode);
159                 }
160                 edge.getTicketIds().add(ticketChangeEntry.getKey());
161 
162             }
163         }
164 
165     }
166 
167     private String toJson() {
168 
169         String jsonString;
170 
171         Map<List<String>, Integer> stateMap = new HashMap<List<String
                >, Integer>();
172         int stateCounter = 0;
173 
174         JSONObject jsonObj = new JSONObject();
175 
176         JSONArray nodeList = new JSONArray();
177         JSONArray linkList = new JSONArray();
178 
179         try {
180             for (SankeyGraphNode node : ticketChangeGraph.vertexSet()
                    ) {
181                 List<String> state = node.getState();
182                 Set<Long> ticketIds = node.getTicketIds();
183                 JSONObject nodeObj = new JSONObject();
184                 if (state.get(state.size() - 1) == null) {
185                     nodeObj.put("name", "null");
186                 } else {
187                     nodeObj.put("name", state.get(state.size() - 1));
188                 }
189                 nodeObj.put("tickets", ticketIds);
190                 nodeList.put(nodeObj);
191 
192                 stateMap.put(state, Integer.valueOf(stateCounter));
193                 stateCounter++;
194             }
195 
196             jsonObj.put("nodes", nodeList);
197 
198             for (SankeyGraphEdge transition : ticketChangeGraph.
                    edgeSet()) {
```

```
199                 JSONObject linkObj = new JSONObject();
200                 linkObj.put("source", stateMap.get(ticketChangeGraph
201                         .getEdgeSource(transition).getState()));
202                 linkObj.put("target", stateMap.get(ticketChangeGraph
203                         .getEdgeTarget(transition).getState()));
204                 linkObj.put("value",
205                         ticketChangeGraph.getEdgeWeight(transition));
206                 linkObj.put("tickets", transition.getTicketIds());
207
208                 linkList.put(linkObj);
209             }
210
211             jsonObj.put("links", linkList);
212             jsonString = jsonObj.toString();
213
214         } catch (JSONException e) {
215             jsonString = "";
216         }
217
218         return jsonString;
219     }
220
221     @Override
222     public String getSankeyJson(String dataSourceIdentifier,
223             String propertyName, List<Filter> filters) {
224
225         calculateChangeGraph(dataSourceIdentifier, propertyName,
                filters);
226         return toJson();
227     }
228
229     @Override
230     public Long getNumberOfTicketsWithChangedProperty(
231             String dataSourceIdentifier, String propertyName,
232             List<Filter> filters) {
233
234         List<TicketChange> filteredTicketChanges =
                getFilteredTicketChanges(
235                     dataSourceIdentifier, propertyName, filters);
236         Set<Long> ticketIds = new HashSet<Long>();
237         for (TicketChange filteredTicketChange :
                filteredTicketChanges) {
238             ticketIds.add(filteredTicketChange.getTicketId());
239         }
240
241         return Long.valueOf(ticketIds.size());
242     }
243 }
```

Quelltext A.2: SankeyCalculatorBean.java

A.4 Ticket-Daten-Modell: SQL-Skript zur Erzeugung des Datenbank-Schemas

```
1  -- -----------------------------------------------------
2  -- DROP tables
3  -- -----------------------------------------------------
4
5  -- disable foreign key contraint checks while dropping tables, so
       drop order does not matter
6  SET foreign_key_checks = 0;
7
8  DROP TABLE IF EXISTS TICKETCHANGE;
9  DROP TABLE IF EXISTS TICKET;
10
11 DROP TABLE IF EXISTS TICKET_PROPERTY;
12
13 -- reactivate foreign key constraint checks
14 SET foreign_key_checks = 1;
15
16 -- -----------------------------------------------------
17 -- Table TICKETCHANGE
18 -- -----------------------------------------------------
19
20 CREATE TABLE TICKETCHANGE (
21   ID INTEGER AUTO_INCREMENT NOT NULL,
22   DATASOURCEIDENTIFIER VARCHAR(255),
23   TICKETID BIGINT,
24   CHANGETIME BIGINT,
25   PROPERTYNAME VARCHAR(255),
26   OLDVALUE VARCHAR(255),
27   NEWVALUE VARCHAR(255),
28   PRIMARY KEY (ID)
29 )
30 ENGINE = InnoDB
31 DEFAULT CHARACTER SET = utf8;
32
33 -- -----------------------------------------------------
34 -- Table TICKET
35 -- -----------------------------------------------------
36
37 CREATE TABLE TICKET (
38   ID INTEGER AUTO_INCREMENT NOT NULL,
39   DATASOURCEIDENTIFIER VARCHAR(255),
40   TICKETID BIGINT,
41   CHANGETIME BIGINT,
42   SUBJECT VARCHAR(255),
43   URL VARCHAR(255),
44   PRIMARY KEY (ID),
45   CONSTRAINT UI_TICKET UNIQUE (DATASOURCEIDENTIFIER, TICKETID)
46 )
47 ENGINE = InnoDB
```

```
48 DEFAULT CHARACTER SET = utf8;
49
50 -- -----------------------------------------------------
51 -- Table TICKET_PROPERTY
52 -- -----------------------------------------------------
53
54 CREATE TABLE TICKET_PROPERTY (
55   TICKET_ID INTEGER NOT NULL,
56   PROPERTYNAME VARCHAR(255) NOT NULL,
57   PROPERTYVALUE VARCHAR(65535) NOT NULL,
58   PRIMARY KEY (TICKET_ID, PROPERTYNAME),
59   CONSTRAINT FK_TICKET_PROPERTY_TICKET_ID
60     FOREIGN KEY (TICKET_ID)
61     REFERENCES TICKET (ID)
62 )
63 ENGINE = InnoDB
64 DEFAULT CHARACTER SET = utf8;
```

Quelltext A.3: SQL-Skript zur Erzeugung des Datenbank-Schemas (createDatabase.sql)

A.5 Ticket-Daten-Verarbeitung: Message-Driven-Bean zum Empfang einer TicketJournalMessage

```
1 [...]
2 @MessageDriven(mappedName = JMSConfig.TOPIC, activationConfig = {
3                 @ActivationConfigProperty(propertyName = "
                      destinationType", propertyValue = "javax.jms.Topic
                      "),
4                 @ActivationConfigProperty(propertyName = "destination
                      ", propertyValue = JMSConfig.TOPIC),
5                 @ActivationConfigProperty(propertyName = "
                      acknowledgeMode", propertyValue = "Auto-
                      acknowledge"),
6                 @ActivationConfigProperty(propertyName = "
                      messageSelector", propertyValue = "messageType = '
                      TicketJournalMessage'") })
7 public class TicketJournalMessageReceiver extends
      AbstractMessageReceiver {
8 [...]
9         @Override
10        public void receiveMessage(BaseGplcrsMessage message) {
11
12                TicketJournalMessage ticketJournalMessage = (
                      TicketJournalMessage) message;
13                List<Ticket> ticketJournal = ticketJournalMessage.
                      getTicketJournal();
14                String dataSourceIdentifier = ticketJournalMessage
15                                .getDataSourceIdentifier();
16                Long ticketId = ticketJournalMessage.getTicketId();
17
18                // clear if exists
```

```
19                  if (ticketDao.ticketExists(dataSourceIdentifier,
                        ticketId)) {
20                      // ticket
21                      Ticket ticketToDelete = ticketDao.
                            getTicketById(
22                                      dataSourceIdentifier,
                                            ticketId);
23                      ticketDao.deleteTicket(ticketToDelete);
24
25                      // and its ticket changes
26                      List<TicketChange> ticketChangesToDelete =
                            ticketChangeDao
27                                      .getAllTicketChangesOfTicket(
                                            dataSourceIdentifier,
                                            ticketId);
28                      for (TicketChange ticketChangeToDelete :
                            ticketChangesToDelete) {
29                          ticketChangeDao.deleteTicketChange(
                                ticketChangeToDelete);
30                      }
31                  }
32
33                  // build ticketchanges
34                  Ticket currentTicketState = null;
35                  for (Ticket newTicketState : ticketJournal) {
36                      if (currentTicketState != null) {
37                          ticketMessageHelper.
                                createTicketChanges(
                                dataSourceIdentifier,
38                                          currentTicketState,
                                                newTicketState);
39                      }
40                      currentTicketState = newTicketState;
41                  }
42
43                  // create a final Ticket State after parsing through
                        the log, that
44                  // is managed by the entity manager
45                  // previousTicketState should never be null, since
                        the DataSourceBeans
46                  // always send at least one ticket in a
                        ticketJournalMessage.
47                  // It could be null due to bugs or if we receive a
                        bogus/malicious
48                  // message
49                  if (currentTicketState != null) {
50                      entityFactory.createTicket(
                            dataSourceIdentifier,
51                                      currentTicketState.
                                            getTicketId(),
52                                      currentTicketState.getSubject
                                            (),
```

```
53                                          currentTicketState.getUrl(),
54                                          currentTicketState.
                                                getChangeTime(),
55                                          currentTicketState.
                                                getProperties());
56                  }
57
58                  // Count the same message once only
59                  if (!isRedelivered()) {
60                          loadProgress.incProcessedCurrentTicket();
61                  }
62
63          }
64  }
```

Quelltext A.4: Auszug aus TicketJournalMessageReceiver.java

A.6 Ticket-Daten-Aktualisierung: XMLRPC-Dienst und Trac-Plugin

```
1  package de.rwth.swc.gplcrs.xmlrpc;
2
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.util.Date;
6  import java.util.Map;
7  import java.util.Properties;
8
9  import javax.jms.JMSException;
10 import javax.naming.InitialContext;
11 import javax.naming.NamingException;
12 import javax.xml.bind.JAXBException;
13
14 import org.apache.commons.lang3.StringUtils;
15 import de.rwth.swc.gplcrs.entity.Ticket;
16 import de.rwth.swc.gplcrs.facade.GplcrsFacadeLocal;
17 import de.rwth.swc.gplcrs.jms.MessageSenderImpl;
18 import de.rwth.swc.gplcrs.jms.message.MessageFactoryLocal;
19
20 public class GplcrsXmlRpcService {
21
22     private static final String EJB_COMMON_MODULE_BASENAME = "ejb-
          common";
23     private static final String EJB_CORE_MODULE_BASENAME = "ejb-core"
          ;
24
25     private GplcrsFacadeLocal facade;
26
27     private MessageSenderImpl messageSender;
28
29     private MessageFactoryLocal messageFactory;
30
```

```
31      private void lookupEjbs() throws IOException, NamingException {
32
33          // load the version from properties file
34          Properties props = new Properties();
35          InputStream is = getClass().getResourceAsStream("xmlrpc.
                properties");
36          try {
37              props.load(is);
38          } finally {
39              is.close();
40          }
41
42          // construct jndi module names
43          String projectVersion = props.getProperty("projectVersion");
44          String ejbCommonModuleName = EJB_COMMON_MODULE_BASENAME + "-"
45                  + projectVersion;
46          String ejbCoreModuleName = EJB_CORE_MODULE_BASENAME + "-"
47                  + projectVersion;
48
49          // lookup session beans in jndi
50          InitialContext ic = new InitialContext();
51          facade = (GplcrsFacadeLocal) ic.lookup("java:app/" +
                ejbCoreModuleName
52                  + "/GplcrsFacadeBean");
53
54          messageSender = (MessageSenderImpl) ic.lookup("java:app/"
55                  + ejbCommonModuleName + "/MessageSenderImpl");
56
57          messageFactory = (MessageFactoryLocal) ic.lookup("java:app/"
58                  + ejbCommonModuleName + "/MessageFactoryBean");
59
60      }
61
62      public String update(String dataSource, Integer ticketId, String
            subject,
63              String url, Date changeTime, Map<String, Object>
                    ticketValues)
64              throws IOException, NamingException, JMSException,
                    JAXBException {
65
66          lookupEjbs();
67
68          final String usage = "Usage: ticket.update(Data Source(string
                ), Ticket Id(int),  Subject(string), Url(string), Change
                Time(dateTime.iso8601), Properties(struct))";
69
70          // Input parameter checking
71          if (dataSource == null) {
72              throw new IllegalArgumentException("Data Source must not
                    be null. "
73                      + usage);
74          }
```

```
75
76          if (ticketId == null) {
77              throw new IllegalArgumentException("Ticket Id must not be
                    null. "
78                      + usage);
79          }
80
81          if (subject == null) {
82              throw new IllegalArgumentException("Subject must not be
                    null. "
83                      + usage);
84          }
85
86          if (url == null) {
87              throw new IllegalArgumentException("Url must not be null.
                    " + usage);
88          }
89
90          if (changeTime == null) {
91              throw new IllegalArgumentException("Change Time must not
                    be null. "
92                      + usage);
93          }
94
95          if (ticketValues == null) {
96              throw new IllegalArgumentException("Properties must not
                    be null. "
97                      + usage);
98          }
99
100         if (!facade.getDataSourceIdentifiers("").contains(dataSource)
                ) {
101             throw new IllegalArgumentException("Data Source " +
                    dataSource
102                     + " does not exist.");
103         }
104
105         Ticket ticket = new Ticket();
106         ticket.setDataSourceIdentifier(dataSource);
107         ticket.setTicketId(Long.valueOf(ticketId.longValue()));
108         ticket.setSubject(subject);
109         if (StringUtils.isNotBlank(url)) {
110             ticket.setUrl(url);
111         }
112         ticket.setChangeTime(Long.valueOf(changeTime.getTime()));
113
114         for (String propertyName : ticketValues.keySet()) {
115             Object propertyValue = ticketValues.get(propertyName);
116             // carry over string properties, only. effectively
                    ignoring Date
117             // properties changetime and time
```

```
118             // if there are other Date properties that are
                    interesting in
119             // analysis, this might pose a problem
120             if (propertyValue instanceof String) {
121                 ticket.setProperty(propertyName, (String)
                        propertyValue);
122             }
123         }
124
125         messageSender.sendMessage(messageFactory.createTicketMessage(
126                 dataSource, ticket));
127
128         return "Ticket " + ticketId + " updated.";
129     }
130 }
```

Quelltext A.5: GplcrsXmlRpcService.java

```
1  # gplcrs plugin
2
3  from trac.core import *
4  from trac.config import Option
5  from trac.util.text import empty
6  from trac.ticket.api import ITicketChangeListener
7
8  import xmlrpclib
9  import socket
10
11 class GplcrsPlugin(Component):
12     implements(ITicketChangeListener)
13
14     GPLCRS_CONFIG_SECTION = "gplcrs"
15
16     gplcrsRestUrl = Option(GPLCRS_CONFIG_SECTION, "rest_url", "",
17         doc="Rest Url of gplcrs. E.g. 'http://localhost:8080/
                gplcrsRest/'. Note the final slash. Required setting.")
18
19     gplcrsDataSource = Option(GPLCRS_CONFIG_SECTION, "data_source", "
           ",
20         doc="Name of the gplcrs Data Source to update with ticket
                change notifications. E.g. 'TRAC - http://localhost:8000/
                trac'. Required setting.")
21
22     def __init__(self):
23         self.baseUrl = self.config.get("trac","base_url")
24
25     # ITicketChangeListener Interface
26     def ticket_created(self, ticket):
27         """Notifies a gplcrs system about a ticket creation"""
28         self.__updateOrCreateTicket(ticket)
29
30     def ticket_changed(self, ticket, comment, author, old_values):
31         """Notifies a gplcrs system about a ticket change"""
```

```
32          self.__updateOrCreateTicket(ticket)
33
34      def ticket_deleted(self, ticket):
35          """NOP when a ticket is deleted."""
36
37      def __updateOrCreateTicket(self, ticket):
38          ticketValues = {}
39          for propertyName in ticket.values.keys():
40              # do not copy empty values, they mean "not set" and are
                    not marshallable
41              if (type(ticket.values[propertyName]) != type(empty)):
42                  ticketValues[propertyName] = ticket.values[
                        propertyName]
43          # construct an url to the ticket, if a base url is set
44          url = ""
45          if (self.baseUrl != ""):
46              url = self.baseUrl + "/ticket/" + str(ticket.id)
47          self.__sendTicketUpdate(ticket.id, ticket.values['summary'],
                url, ticket.values['changetime'],  ticketValues)
48
49      def __sendTicketUpdate(self, id, subject, url, changetime, values
            ):
50          """sends a ticket update to gplcrs XML RPC"""
51
52          if (self.gplcrsRestUrl == ""):
53              self.log.warn("option rest_url not set")
54              return
55          if (self.gplcrsDataSource == ""):
56              self.log.warn("option data source not set")
57              return
58          xmlRpcServer = xmlrpclib.ServerProxy(self.gplcrsRestUrl,
                use_datetime=1, allow_none=1)
59          try:
60              rpcResult = xmlRpcServer.ticket.update(self.
                    gplcrsDataSource, id, subject, url, changetime, values
                    )
61              self.log.debug("rpcResult = " + rpcResult)
62          except socket.error, (value,message):
63              self.log.error(str(value) + ": " + message);
64          except xmlrpclib.Fault as err:
65              self.log.error("A fault occurred")
66              self.log.error("Fault code: %d", err.faultCode)
67              self.log.error("Fault string: %s", err.faultString)
68          except xmlrpclib.ProtocolError as err:
69              self.log.error("A protocol error occurred")
70              self.log.error("URL: %s", err.url)
71              self.log.error("HTTP/HTTPS headers: %s", err.headers)
72              self.log.error("Error code: %d", err.errcode)
73              self.log.error("Error message: %s", err.errmsg)
```

Quelltext A.6: Trac-Plugin gplcrs.py

Literaturverzeichnis

[Abr02] Abrahamsson, P. and Salo, O. and Ronkainen, J. and Warsta, J. *Agile software development methods. Review and analysis.* Espoo 2002, VTT Publications 478, 2002.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[BCR94] V. Basili, G. Caldiera, and H.D. Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*, pages 528–532. Wiley, 1994.

[Cha05] Robert N. Charette. Why Software Fails. `http://spectrum.ieee.org/computing/software/why-software-fails`, September 2005. Zugriffsdatum: 20.04.2013.

[CJ02] R.D.A. CRAIG and S.P. Jaskiel. *Systematic software testing.* Artech House computer science library. ARTECH HOUSE Incorporated, 2002.

[CVW98] Jonathan E. Cook, Lawrence G. Votta, and Alexander L. Wolf. Cost-effective analysis of in-place software processes. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 24(8):650–663, 1998.

[DeM86] T. DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[DeM09] Tom DeMarco. Software engineering: An idea whose time has come and gone? *IEEE Software*, 26(4):96–95, 2009.

[Eme12] Elena Emelyanova. Regelbasierte Initialisierung von Projektdashboards. Bachelor thesis, RWTH Aachen University, 2012.

[ET94] Peter Eades and Roberto Tamassia. Algorithms for drawing graphs: An annotated bibliography. Technical report, Brown University, Providence, RI, USA, 1994.

[Eve12] Frederic Evers. Konzeptionelle Erweiterung von Projektdashboards für unerfahrene Anwender. Master thesis, RWTH Aachen University, 2012.

[Fö97] Fößmeier, U. *Orthogonale Visualierungstechniken für Graphen.* Dissertation, Fakultät für Informatik, Eberhard-Karls-Universität zu Tübingen, 1997.

[Few06] Stephen Few. *Information Dashboard Design: The Effective Visual Communication of Data.* O'Reilly Media, Inc., 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gji13] Endri Gjino. Visually Assisted Mining for Smells in Change Request Systems. Master thesis (noch unveröffentlicht), RWTH Aachen University, 2013.

[Gor13] Matthias Gora. Entwicklung eines Dashboard Prototyping-Werkzeugs. Bachelor thesis, RWTH Aachen University, 2013.

[Han12] Christian Hans. Einsatz von Metrik-Dashboards im industriellen Umfeld. Bachelor thesis, RWTH Aachen University, 2012.

[IEE90] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, December 1990.

[ISO07] ISO. Systems and software engineering — Measurement process. Norm ISO/IEC 15939:2007, International Organization for Standardization, Geneva, Switzerland, 2007.

[ISO10] ISO. Systems and software engineering — Vocabulary. Norm ISO/IEC 24765:2010, International Organization for Standardization, Geneva, Switzerland, 2010.

[Joh01] Philip Johnson. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. `http://www.nitrd.gov/nitrdgroups/images/3/30/Philip_johnson_you_cant_even_ask.pdf`, October 2001. Zugriffsdatum: 23.04.2013.

[Lic11] Horst Lichter. Vorlesungsunterlagen: Software Qualitätssicherung, 2011.

[Lic12] Horst Lichter. Vorlesungsunterlagen: Software Project Management, 2012.

[LL10] Jochen Ludewig and Horst Lichter. *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, Heidelberg, 2010.

[Mä13] Martin Mädler. Variabilität von Metriken und Dashboard-Items im Umfeld von MeDIC. Diploma thesis, RWTH Aachen University, 2013.

[May08] Horst O. Mayer. *Interview und schriftliche Befragung: Entwicklung, Durchführung und Auswertung*. Oldenbourg Verlag, München, 4. edition, 2008.

[Rum10] Bernhard Rumpe. Vorlesungsunterlagen: Einführung in die Softwaretechnik, 2010.

[Sch06] Mario Schmidt. *Der Einsatz von Sankey-Diagrammen im Stoffstrommanagement*. Number 124 in Beiträge der Hochschule Pforzheim. Hochsch., Pforzheim, 2006.

[Ste13] Andreas Steffens. Entwurf eines Architekturmodells zur Integration heterogener Systeme in MeDIC . Diploma thesis, RWTH Aachen University, 2013.

[Via12] Matthias Vianden. MeDIC - Eine Infrastruktur zum Verwalten, Dokumentieren und Visualisieren von Metriken. In *Entwicklung und Evolution von Forschungssoftware, Tagungsband, Rolduc; [10.-11.11. 2011] / Rumpe, Bernhard [Hrsg.] ; Lichter, Horst [Hrsg.]*, Aachener Informatik Berichte Software Engineering, Band 14, pages 19–22, Herzogenrath, 2012. Shaker Verlag.