2 Grundlagen

Inhaltsangabe

2.1 Qualität 3
2.2 Metriken 5
2.3 Change-Request-Systeme 12
2.4 Sankey Diagramme 12
2.5 Java EE 15
2.6 Weitere Technologien 16

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.

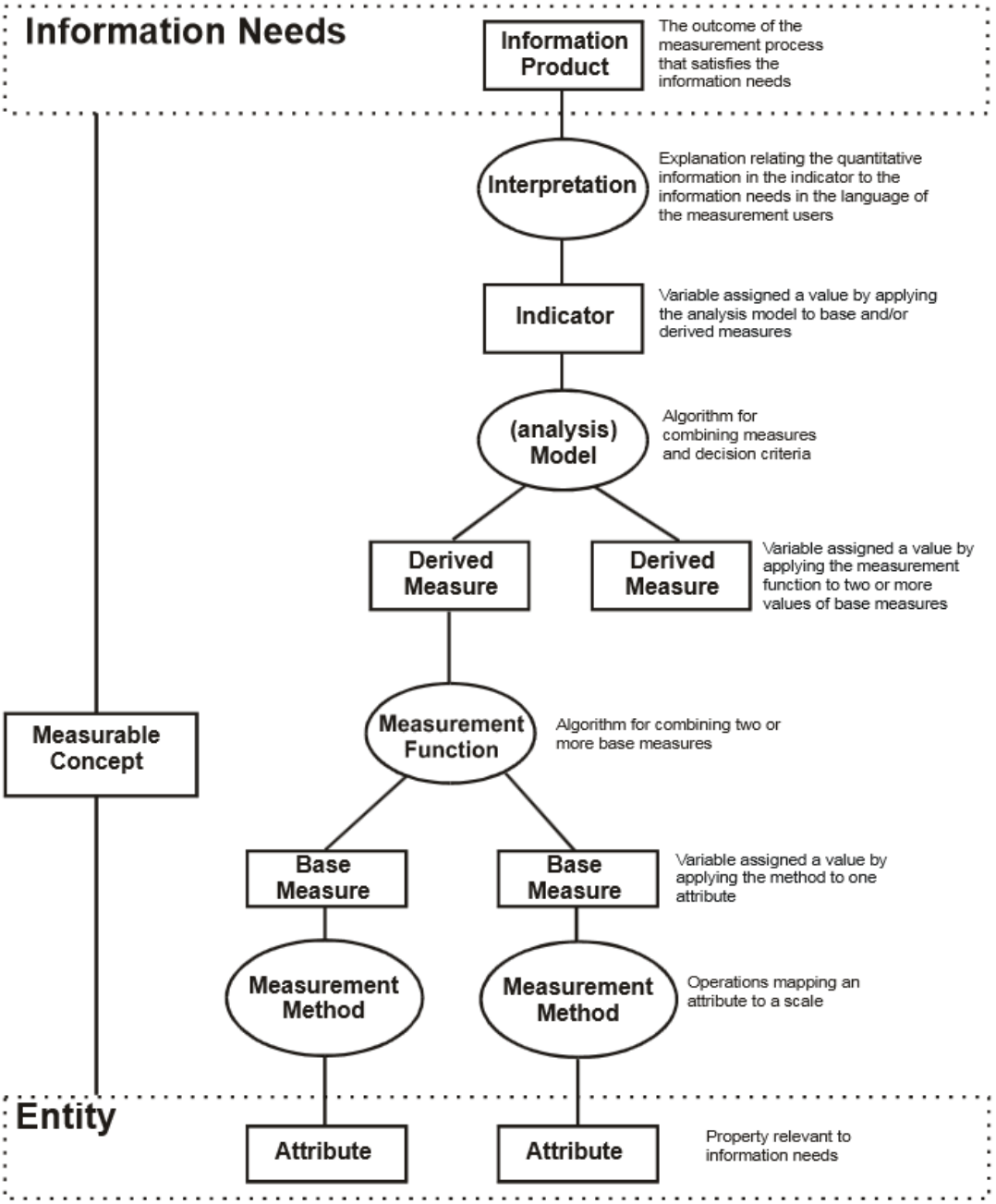

Abbildung 2.1: Maß-Informations-Modell nach ISO/IEC 15939

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* ...

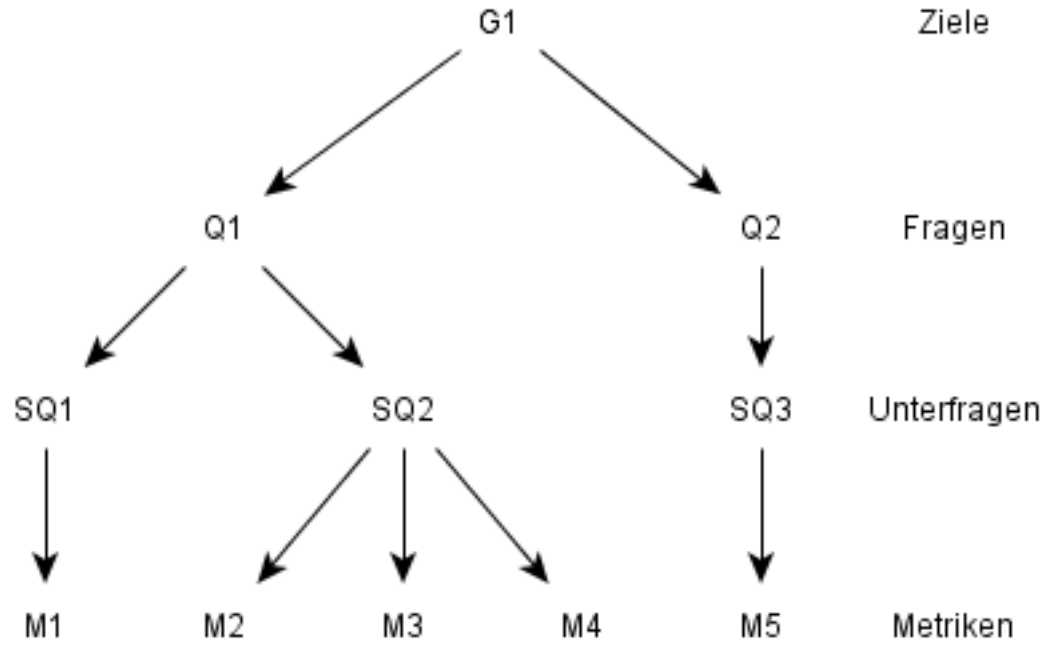

Abbildung 2.2: Goal-Question-Metric-Ansatz

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.

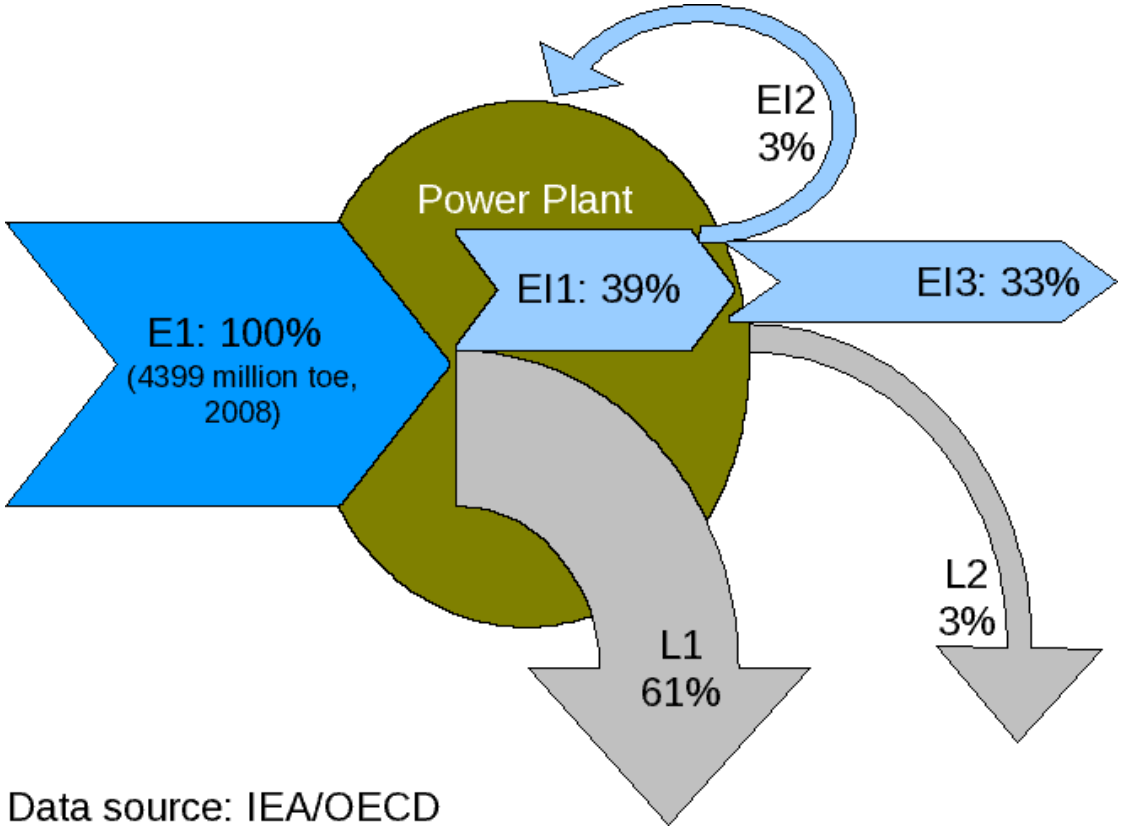

Abbildung 2.3: Energiefluss-Sankey-Diagramm
Quelle: `http://de.wikipedia.org/wiki/Sankey-Diagramm`, abgerufen am 10.06.2013

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.