

Technische Universität München
Fakultät für Informatik

Hauptseminar

Management von Softwaresystemen
Thema: Systembewertung: Metriken und Prozess

Referent: Vadym Alyokhin
Betreuer: Florian Deißeböck

INHALTSVERZEICHNIS

1	EINLEITUNG	1
2	HISTORIE VON SOFTWARE-METRIKEN	3
2.1	Ursprung der Software-Metrik	3
2.2	Meilensteine von Software-Metriken	4
3	SYSTEMATISCHE ANWENDUNG VON METRIKEN	7
3.1	Einführung in die Messtheorie	7
3.2	Klassifikation von Software-Metriken	9
3.3	Goal-Question-Metric Paradigma	11
3.4	Kritik an Software-Messung	12
4	SOFTWARE-METRIKEN UND REENGINEERING	14
4.1	Technische Bewertung	16
4.2	Ökonomische Bewertung	19
4.3	Management Prozess	21
5	ZUSAMMENFASSUNG	23
	ABKÜRZUNGEN UND BEGRIFFE	24
	TABELLENVERZEICHNIS	26
	ABBILDUNGSVERZEICHNIS	27
	LITERATURVERZEICHNIS	28

1 Einleitung

Not everything that counts can be counted
and not everything that is counted counts.
Albert Einstein

Die Software-Metriken sind ohne Zweifel eine Beliebtheit. Es gibt sie seit den ersten programmierbaren Maschinen. Besonders interessant war damals die Größe der benötigten Speicher, oder – direkt damit zusammenhängend – die Zahl der benötigten Lochkarten. Heute wird überall gemessen und verglichen. Fehlerzahlen und Fehlerraten bestimmen über Produktfreigaben. Kunden erfragen die neuesten Prozessmetriken, um sich ein Bild über die Prozessqualität in der Produktentstehung machen zu können. Manager vergleichen Produktivitätszahlen und den „Return on Investment“ von neuen Methoden oder Werkzeugen.

Ziel von Software-Messung ist es natürlich nicht, einfach zu messen, was messbar scheint, sondern ganz im Sinne von A. Einstein, was wirklich zählt. Daher zielt die Disziplin auch hin zu reproduzierbaren Experimenten, definierten Umgebungen und will damit Vorgehensweisen und Resultate schaffen, die sich draußen umsetzen und anwenden lassen.

Der Einsatz von Metriken wird in der Praxis allgemein als nötige Voraussetzung zur Beherrschung der Prozesse bei der Erstellung, Handhabung und Pflege der zunehmend komplexeren Software anerkannt. Prinzipiell sind Software-Metriken unumstritten.

Aber wie sollte man, oder wie kann man damit beginnen? Welcher Ansatz einer Software-Messung führt möglichst gleich zu sichtbaren Ergebnissen? Welche Metriken existieren? Wie kann man die Metriken systematisch einsetzen? Was wäre eine typische Anwendung für die Metriken?

Die vorliegende Arbeit versucht die Fragen zu beantworten. Sie versucht eine Orientierung zu geben und verweist auf die weiterführende und vertiefende Literatur. Als Beispiel der Anwendung für die Software-Metriken wird der „Software Reengineering Assessment Process“ von Department of Defense (DoD) vorgestellt. Dieser Prozess beschreibt eine Reihe von technischen und ökonomischen Entscheidungen und Managementaktivitäten für ein Reengineering-Bewertungsprozess.

Daher besteht die Arbeit aus zwei Teilen. Der erste Teil gibt einen Überblick über die bekanntesten Metriken, relevante Messtheoretische Ansätze und das systematische Vorgehen für die Anwendung von Metriken. Dieser Teil wird mit einem Fallbeispiel und Einschränkungen für Software-Messung abgerundet. Der zweite Teil ist den Metriken im Reengineering-Kontext gewidmet. Das zentrale Thema ist dabei die Ausarbeitung der wichtigsten Punkte des „Software Reengineering Assessment Handbook“.

Nach der Einleitung im Kapitel 2 werden die Software-Metriken mit einem historischen Hintergrund betrachtet. Der Abschnitt 2.1 „Ursprung der Software-Metrik“ erzählt beispielhaft darüber, wie das Messen entstanden ist. Der nächste Abschnitt 2.2 „Meilensteine von Software-Metriken“ gibt einen chronologischen Überblick über die wichtigsten Entwicklungen und einige Namen in der Software-Messung.

Antwort auf die Frage, wie kann man effektiv die Software-Metriken einsetzen, bekommt man im Kapitel 3. Der Abschnitt 3.1 ist eine grundlegende Einführung in die Messtheorie. Dies soll eine Basis für den Einsatz aller Metriken geben. Im Abschnitt 3.2 wird auf Klassifikation der Software-Metriken eingegangen. Ein systematischer Ansatz für die Ableitung der Software-Metriken - „Goal-Question-Metric (GQM) Approach“- ist im Abschnitt 3.3 dargestellt. Kann man alles mit den Software-Metriken messen? Mit dieser Frage wird man im Abschnitt 3.4 konfrontiert.

Das Kapitel 4, angefangen mit Grundlagen des Reengineering, gibt eine Einführung in das „Software Reengineering Assessment Handbook“, ein grundlegender Standard der DoD über Richtlinien für die Analyse existierender Software, mit dem Ziel der Wiederverwendung. Es wird in drei Bereiche eingeteilt: die technische und ökonomische Bewertung und der Managementprozess. Die Bereiche

entsprechen den Abschnitten 4.1, 4.2 und 4.3. Die technische Bewertung besteht aus 6 Schritten und basiert auf Fragenkatalogen. Sie identifiziert Software-Komponenten für Reengineering und schlägt Vorgehensstrategien vor („Redocumentation“, „Reverse Engineering“, „Restructuring“ usw.). Die ökonomische Bewertung basiert auf der Metriken, anhand derer aus den vorgeschlagenen technischen Bewertungsstrategien die effizienteste ausgewählt wird. Der Managementprozess beschäftigt sich dabei mit begleitenden Aufgaben: Report, Ressourcenanalyse und Dokumentation.

2 Historie von Software-Metriken

Dieses Kapitel führt Motivationen ein, beschreibt Praktikabilität eines Maßes in der Maß-Geschichte und die wichtigsten Ereignisse in der Software-Metrik-Geschichte. Der Abschnitt 2.1 gibt einen Überblick über die Entwicklung von Maßen und die Entstehung von Software-Metriken. Der nächste Abschnitt 2.2 geht auf die Geschichte der Software-Metrik ein.

2.1 Ursprung der Software-Metrik

Das *Maß* gehört zu den frühesten Werkzeugen, das die Menschheit erfunden hat. Das alltägliche Leben von Menschen ist unvorstellbar ohne solche Einheiten, die ein gemeinsames System für Verständnis und eine Grundlage für alle Wissenschaften bilden. Beim Anfang des Handels, der Produktion von Alltagswaren wie Kleidung oder bei der Entstehung von Bautätigkeiten entwickelten sich die ersten Maße. Als Basis für die Maße haben sich oft die Umgebungseigenschaften oder verschiedene Körperteile durchgesetzt. Einige davon werden sogar bis heute unverändert gebraucht, zum Beispiel die Gewichtseinheit für Schmucksteine – Karat – wurde vom Samen des Johannisbrotbaums abgeleitet.

Je mehr von Menschen in der Naturwissenschaft erforscht wurde, desto mehr entstand die Notwendigkeit für übersichtliche und nützliche Darstellung von neuer Information. Dabei muss die Information nicht nur von anderen korrekt interpretiert werden können, sondern auch vergleichbar sein. Die Anforderung an die Vergleichbarkeit hat sich als sehr wichtig in der Entwicklung der Wissenschaft herausgestellt. So führte es im Verlaufe des 18. Jahrhunderts in Europa zur Entstehung des auf dem Meter basierenden *Einheitssystem*s. Das höhere Ziel für die Gültigkeit des metrischen Systems lautet: „Für alle Welt, für alle Völker“. Jedes Land, das den internationalen Handel, sowie wissenschaftlichen und technischen Austausch unterstützen wollte, musste das System akzeptieren.

Mit der Entwicklung der Software-Engineering-Disziplin wurde ziemlich früh die Notwendigkeit nach vergleichbaren Maßen erkannt. Seit mehr als dreißig Jahren wurden viele solcher Maße erfunden. Der eingeprägte Name für ein Maß in der Software-Technik ist *Metrik*, wobei die Metrik nicht das gleiche ist, wie die Metrik in der Mathematik. Der Unterschied ist, dass die mathematische Metrik ein Abstandsmaß definiert.

Was ist eigentlich eine Software-Metrik? Der IEEE-Standard 1061 von 1992 definiert eine Softwarequalitätsmetrik wie folgt: „Eine Softwarequalitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit“. Mit anderen Worten wird die Software-Qualität linear auf Software-Metrik abgebildet. In [Grady 1992] wurde über Software-Metriken geschrieben: „We use software measures to derive:

- A basis for estimates,
- To track project progress,
- To determine (relative) complexity,
- To help us to understand when we have achieved a desired state of quality,
- To analyze or defects,
- And to experimentally validate best practices,

in short: they help us to make better decision”.

Die Standardisierung ist eine Voraussetzung für die Akzeptanz und die Langlebigkeit einer Metrik. Heutzutage kennt man verschiedene technische Normen, die Eigenschaften von technischen Systemen, technische Schnittstellen, Prozesse und Messverfahren festlegen. Zum Beispiel, in Deutschland

herrschen die Normen mit dem Kürzel DIN. Das Kürzel EN sagt, dass die entsprechende Norm europäischer Beschluss ist, ISO – internationaler. Viele Metriken aus den Bereichen Informatik sind standardisiert und finden ihre Anwendung in genormten Prozessen. Dazu zählen, zum Beispiel, die Metriken der „Function Point“ Analyse und die Überdeckungskriterien-Metriken. Die Standardisierung von Software-Metriken ist bestimmt ein wichtiges Thema, weil sie ein Austausch von Information nicht nur von einem Projekt zum anderen ermöglicht, sondern auch zwischen Unternehmen. Das Thema wird hier aber nicht weiter vertieft, weil es keinen Schwerpunkt des Artikels darstellt.

2.2 Meilensteine von Software-Metriken

Die erste Software-Metrik, die sich durchgesetzt hat, ist die Metrik „*Lines of Code*“ (LOC), die noch bis heute benutzt wird [Park 1992]. Die Metrik wurde für unterschiedliche Zwecke verwendet. Im Jahr 1974 hat Wolverton [Wolverton 1974] die ersten Versuche gemacht, um mit Hilfe von LOC die Produktivität von Programmierern zu messen. Die Idee der Messung LOC, aus [Shepperd 1993] ist, dass die Programmlänge für die Voraussage solcher Charakteristika, wie der Zuverlässigkeit und der Komplexität der Wartung benutzt werden kann. Die Autoren von [Walston 1977] haben vorgeschlagen, die Produktivität mit Hilfe der voraussichtlichen und aktuellen Anzahl von LOC zu messen. Insgesamt wurden über diese Metrik mehr als tausend Publikationen geschrieben. Wegen ihre Banalität wurde die Metrik sehr oft kritisiert. Für viele Aussagen, die zum Beispiel von der Struktur des Codes abhängen, kann diese Metrik als „Null Hypothese“ benutzt werden, aber eine definitive Entscheidung muss anhand von anderen effektiven Metriken getroffen werden [Zuse 1998].

Die erste Publikation über *Software-Komplexität* erschien im Jahre 1968 von [Rubey 1968]. 1979 schrieb Belady [Belady 1979] eine Dissertation über Software-Komplexität. Dabei wurden zum ersten Mal folgende Fragen diskutiert: Was ist Software-Komplexität? Ist die Software-Komplexität quantitativ messbar?

In den sechziger Jahren wurden die ersten *Kostenschätz-Methoden* Delphi und Nelson's SDC vorgestellt [Helmer-Heidelberg 1966] [Nelson 1966].

Die bekanntesten Metriken, die bis heute noch aktiv diskutiert werden, die McCabe [McCabe 1976] und Halstead-Metrik [Halstead 1977] erschienen Mitte der siebziger Jahre. *McCabe* leitete seine Metrik für Software-Komplexität aus der Graphen-Theorie ab und benutzte dabei die Definition der zyklomatischen Zahl. Er interpretierte die zyklomatische Zahl als die minimale Anzahl von Pfaden in einem Kontrollflussgraph. Die Metrik wird berechnet als

$$v(G) = e - n + 2p$$

wobei e die Anzahl der Kanten im Graphen, n die Anzahl der Knoten im Graphen und p die Anzahl der verbundenen Komponenten ist. Unter einer verbundenen Komponente versteht man einen einzelnen Kontrollflussgraphen. Besteht ein Programm aus mehreren Methoden, so wird jede Methode als eigener Kontrollflussgraph dargestellt. In Abbildung 1 ist ein Beispiel der Metrik dargestellt.

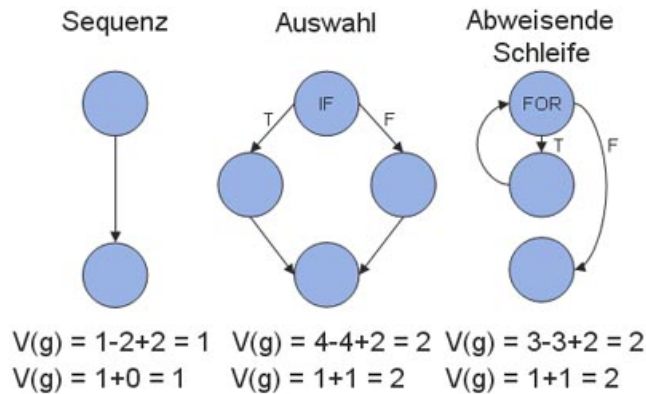


Abbildung 1: Berechnung der McCabe-Metrik

Die *Messung von Halstead* basiert auf dem Quellcode eines Programms. Halstead zeigt, dass der voraussichtliche Aufwand, oder die Programmierungs-Zeit, als eine Funktion von der Anzahl der Operatoren und Operanden dargestellt werden kann. Die Metrik wird heute beispielsweise für das Messen von Programmlänge, Programmumfang, Schwierigkeitsgrad und Programmieraufwand verwendet.

- Programmlänge: $N = N_1 + N_2$
- Programmumfang: $V = N * \log_2(n_1 + n_2)$
- Schwierigkeitsgrad:
$$D = \frac{n_1 * N_2}{2n_2}$$
- Programmieraufwand: $E = D * V$

wobei

- N_1, N_2 sind die Gesamtzahlen verwendeter Operatoren, Operanden
- n_1, n_2 sind die Anzahl unterschiedlichen Operatoren, Operanden

Im Jahr 1977 haben Laemmel und Shooman [Laemmel 1977] anhand von „Zipf's Law“, der für natürliche Sprachen entworfen war, eine erweiterte Theorie für Programmiersprachen analysiert. „Zipf's Law“ ist eine Komplexitätsberechnung, die auf Operatoren, Operanden und die Kombinationen zwischen Operatoren und Operanden basiert. Das Ergebnis dieser Arbeit hat gezeigt, dass das Gesetz sich für Programmiersprachen eignet und sehr ähnlich den Halstead-Metriken ist.

Im gleichen Jahr wurden zwei weitere Software-Komplexitätsmetriken publiziert: „Interval-Derived-Sequence-Length“ (IDSL) und „Loop-Connectedness“ (LC) [Hecht 1977]. Im Jahr 1978 wurde eine Komplexitätsmetrik von McClure veröffentlicht [McClure 1978].

Albrecht [Albrecht 1979] hat 1979 die „Function-Point“ Methode zur Messung von Entwicklungs-Produktivität eingeführt.

Im Jahr 1980 hat Oviedo [Oviedo 1980] das „Model of Program Quality“ entwickelt. Das Modell definiert die Komplexität eines Programms anhand von Kontrollfluss- und Datenfluss-Komplexität in einer Metrik.

Im Jahr 1981 hat Ruston [Ruston 1981] die Messung der Komplexität mit Hilfe von Kontrollfluss-Elementen und deren Struktur vorgeschlagen.

Troy hat 1981 [Troy 1981] eine Menge aus 24 Metriken für die Analyse der Modularität, der Größe, der Komplexität, der Kohäsion und der Kopplung von Software-Systemen eingeführt. Insbesondere

hat die Messung von Kohäsion und Kopplung eine große Verbreitung erreicht. Über eine weitere Entwicklung von Kohäsions- und Kopplungs-Metriken, sowie über Änderungen die andere Metriken überlebt haben, kann in [Zuse 1998] nachgeschlagen werden.

Im Jahr 1981 hat Boehm [Boehm 1981] das COCOMO (Constructive Cost Model) vorgestellt. Das Modell basiert auf der LOC-Metrik und besteht aus vier Methoden für die Kosten-Einschätzung: Experten-Meinung, Analogie, Dekomposition und Gleichungen für die Einschätzung. Eine gute Übersicht über das Modell ist in [Fenton 1991] dargestellt.

Ende der achtziger Jahre, mit dem steigenden Interesse an der Objekt-Orientierten Programmierung, hat sich der Bedarf nach entsprechenden Metriken ergeben. 1989 [Morris 1989] wurden die Software-Metriken für die Anwendung in Objekt-Orientierten Applikationen diskutiert. 1994 wurde das erste Buch über Objekt-Orientierte Metriken geschrieben.

Ein wichtiges Ereignis in der Welt der Objekt-Orientierung ist die Beschreibung einer Gruppe von Objekt-Orientierten Metriken im Jahr 1994 von [Chidamber 1994]. Die Gruppe besteht aus sechs Metriken:

- „Weighted Method Count“ – charakterisiert den Umfang eines Programms als die Summe der Komplexitäten aller Methoden einer Klasse
- „Coupling Between Object Classes“ – charakterisiert die Interaktionskopplung einer Klasse
- „Depth of Inheritance Tree“ ist ein Maß der Vererbungskopplung; es wird durch die Tiefe der Vererbungshierarchie ausgedrückt
- „Number of Children“ ist auch ein Maß der Vererbungskopplung; es wird durch die Anzahl der abgeleiteten Klassen ausgedrückt
- „Response For a Class“ - charakterisiert Interaktionskopplung durch die Anzahl von Methoden, die auf eine Nachricht reagieren
- „Lack of Cohesion in Methods“ – beschreibt Kohäsion der Methoden in einer Klasse

1992 wurde der erste IEEE-Standard über Qualitäts-Metriken - „Software Quality Metric Standard IEEE 1061“ publiziert. Dieser Standard beschreibt Methoden für die Ermittlung von Qualitätsanforderungen, Identifikation, Analyse und Validierung von Qualitäts-Metriken. Unter den für die Software-Messung wichtigen Standards, muss man unbedingt den internationale „Software-Qualität Standard ISO 9000“ und den „Software Product Evaluation Standard ISO 9126“ nennen.

3 Systematische Anwendung von Metriken

In diesem Kapitel werden als erstes die messtheoretischen Ansätze vorgestellt. Es wird gezeigt, welchen Bedingungen die Metriken genügen müssen, um eine aussagekräftige Basis für Software-Messungen zu bilden. Weiter in Abschnitt 3.2 werden zwei verschiedene Klassifikations-Ansätze für Software-Metriken vorgestellt: nach dem Software-Entwicklungsprozess und nach Metrik-Eigenschaften. Abschnitt 3.3 zeigt ein Beispiel für systematische Definition von Prozess-Metriken – „Goal-Question-Metric Paradigma“. In dem abschließenden Abschnitt der Kapitel wird auf einige Probleme der Software-Messung eingegangen und ein Evaluierungs-Framework für Metriken vorgestellt.

3.1 Einführung in die Messtheorie

Software Messung wie die Messung in allen anderen Disziplinen muss sich auf einer wissenschaftlichen Basis aufbauen. Nur auf diese Weise wird die gemeinsame Akzeptanz und Validität erreicht. Das Übersehen von sehr einfachen, aber fundamentalen Prinzipien der Messung kann extreme negative Konsequenzen für das erzielte Ergebnis haben [Fenton 1994].

Was ist genau die *Messung*? Laut [Finkelstein 1984]: „**Measurement** is defined as the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules“. Eine Einheit kann ein Objekt, ein Mensch, eine Spezifikation, ein Ereignis oder eine Test-Phase eines Software-Projektes sein. Ein Attribut ist ein Merkmal oder eine Eigenschaft der Einheit, zum Beispiel der Umfang einer Spezifikation, die Kosten einer Entwicklung oder der Dauer der Test-Phase.

In der Messtheorie müssen gewisse Präzisionen gelten. Die Abbildung der Zahlen oder Symbolen muss die intuitive oder empirische Vorstellung über die Attribute und Einheiten beibehalten. Zum Beispiel bei einer Größen-Messung eines Menschen müssen die höheren Messwerte auf größere Menschen zuweisen. Das Attribut „Größe eines Menschen“ wird aber unterschiedlich interpretiert, abhängig davon ob die Frisur oder die Höhe der Schuhe auch mitgemessen wird. Die Lösung des Problems liegt in der Definition eines *Modells*, das von einem Blickwinkel betrachtet wird. In unserem Beispiel kann das Modell über die Körperhaltung, Frisur und Schuhe bei der Größen-Messung gewisse Regeln definieren [Fenton 1994].

Ein gutes Modell ist die Voraussetzung für die Software Engineering Messungen. Wenn man zum Beispiel die einfachste Metrik Lines of Code (LOC) betrachtet, können die numerischen Werte sehr unterschiedlich interpretiert werden. Das hängt zum Beispiel von der verwendeten Programmiersprache, dem Programmierstil, von der Berücksichtigung von Kommentaren usw. ab. Deswegen muss, abhängig von dem Anwendungs-Kontext, ein geeignetes Modell definiert werden.

Für eine Messung mit dem Ziel der Vorhersage reicht das oben beschriebene Modell nicht aus. Dafür werden zusätzlich die Verfahren gebraucht für a) Ermittlung der Modell-Parameter und b) Interpretation des Ergebnisses. Zum Beispiel für die Zuverlässigkeitsabschätzung mit „Maximum Likelihood“ Verfahren können unterschiedliche Zuverlässigkeitsmodelle wie „Musa“ oder „logarithmisches Modell“ eingesetzt werden, die auch unterschiedliche Ergebnisse liefern.

Im Weiteren wird auf die wichtigsten Punkte der repräsentativen Messtheorie eingegangen.

Empirisches Relationssystem. Bevor man mit der direkten Messung der einzelnen Attribute von Einheiten beginnt, müssen die Attribute intuitiv verstanden werden. Dieses intuitive Verständnis führt zur Definition einer empirischen Relation zwischen Einheiten. Die Menge der Einheiten C , zusammen mit der empirischen Relation R , ist als empirisches Relationssystem (C, R) für ein Attribut genannt. Zum Beispiel das Attribut „Größe eines Menschen“ mit der empirischen Relation wie „ist größer“, „größer als“ und „viel größer als“ bilden ein empirisches Relationssystem.

Repräsentations-Bedingungen. Um ein Attribut eines empirischen Relationssystems (C, R) zu messen, wird eine Abbildung M in das numerische Relationssystem (N, P) gebraucht. Bei der Abbildung M werden die Einheiten in C zu den Zahlen (oder Symbolen) in N , und empirische Relation in R zu numerischer Relation in P abgebildet. Das ist eine so genannte Repräsentations-Bedingung und die Abbildung M ist die Repräsentation. Die Repräsentations-Bedingung stellt sicher, dass M eine homomorphe Abbildung darstellt.

Zum Beispiel: C ist die Menge aller Leute und R enthält die Relation „größer als“. Die Messung M bildet C in die Menge der reellen Zahlen \mathfrak{R} ab. Und die Relation „größer als“ wird zu der Relation „>“ abgebildet. Die Repräsentations-Bedingung stellt sicher, dass eine Person A größer ist als eine Person B , dann und nur dann wenn $M(A) > M(B)$.

Skala-Typen und deren Interpretation. Im Allgemeinen gibt es unterschiedliche Wege für die Bestimmung der Zahlen, die die Repräsentations-Bedingung erfüllen. Zum Beispiel, wenn Person A größer als Person B ist, dann ist $M(A) > M(B)$ unabhängig davon, ob M in Zoll oder Zentimeter ausgedrückt wird. Die Transformation von einer gültigen Darstellung zur anderen heißt *zulässige Transformation*. Zum Beispiel $M = c * M'$, wobei M die Darstellung der Größe in Zoll, M' in Zentimeter ist, und $c = 2,54$.

Die zulässigen Transformationen bestimmen ein Skala-Typ für ein Attribut. Zum Beispiel, wenn alle zulässigen Transformationen skalare Multiplikationen (wie im Falle der Höhe-Messung) sind, dann wird der Skala-Typ *Rationalskala* genannt. Nicht jedes Attribut ist a priori vom Rationalskala-Typ. An der Stelle kann an das Attribut „Kritikalität“ erinnert werden. Das empirische Relationssystem wird definiert, zum Beispiel, als verschiedene Klassen von Fehlern und die Relation „ist mehr kritisch als“. In dem Fall stehen zwei beliebige Darstellungen aus der Relation in einer monoton wachsenden Beziehung zu einander. Mit dieser Klasse von zulässigen Transformationen wird die *Ordinalskala* assoziiert. Nach der Verfeinerungs-Ordnung können die Skala-Typen wie folgt aufgezählt werden: Nominalskala, Ordinalskala, Intervallskala, Rationalskala und Absolutskala. Die Eigenschaften von Skalen sind in der Tabelle 1 dargestellt. Das Thema kann weiter in [Roberts 1979] vertieft werden.

Die formale Definition des Skala-Typs, basierend auf eine zulässige Transformation, hilft festzustellen, welche Art von Aussagen über die Messung erlaubt sind. Zum Beispiel, wenn die Aussage „Florian doppelt so groß wie Peter“ bei der Messung in Zentimeter wahr ist, dann bleibt sie auch wahr, wenn die Messung in Zoll erfolgt. Anderes betrachtet: die Aussage „Temperatur x ist doppelt so hoch wie Temperatur y “ ist nicht aussagekräftig in der Ordinal-Skala eines empirischen Relationssystems für die Temperatur. Eine gültige Ordinal-Skala Messung M der Temperatur in Fahrenheit kann ergeben $M(x) = 64\text{F}$, $M(y) = 32\text{F}$, obwohl die andere Ordinal-Skala Messung M' der Temperatur in Celsius $M'(x) = 18^\circ\text{C}$ und $M'(y) = 0^\circ\text{C}$ ergibt. Nur in der rationalen Skala bekommt die Aussage einen Sinn, zum Beispiel, „Temperatur 200 Kelvin ist doppelt so hoch wie 100 Kelvin“.

Skalen	Eigenschaften	Erlaubte Operationen	Statistische Maß	Beispiel
Nominalskala	Reine Kategorisierung von Werten	=, ≠	Modus	Blau, grün, rot
Ordinalskala	Skalenwerte geordnet und vergleichbar	=, ≠, >, <	Median	Sehr kalt, kalt, warm, heiß, sehr heiß
Intervallskala	Werte geordnet, Distanzen bestimmbar	=, ≠, >, <, +, -	Mittelwert	20°C
Rationalskala	Werte geordnet. Skala hat einen absoluten Nullpunkt	=, ≠, >, <, +, -, *, ÷, %	Mittelwert	273°K
Absolutskala	Skalenwerte sind absolute Größen	=, ≠, >, <, +, -, *, ÷, %	Mittelwert	Aufzählungen

Tabelle 1: Übersicht über verschiedene Maßskalen.

3.2 Klassifikation von Software-Metriken

Es existieren mehrere Klassifikationen von Software-Metriken. Allgemein kann man zwischen Produkt- und Prozessmetriken unterscheiden. Wobei es unter den Produktmetriken, Metriken statischer und dynamischer Natur gibt. Somit ergibt sich die in der Abbildung 2 vorgestellte Taxonomie.

Eine erfolgreiche Methode Metriken zu klassifizieren ist die Klassifikation nach dem Software-Engineering Entwicklungsprozess. Eine übersichtliche Darstellung davon mit typischen Kenngrößen ist in der folgenden Abbildung 3 präsentiert.

[Balzert 1998] benutzt folgende Kriterien, die teilweise orthogonal zu einander stehen, für eine Klassifikation von Metriken.

Objektiv/Subjektiv. Zu *objektiven Metriken* gehören die Metriken, die leicht quantifizierbar und messbar sind, z.B. Programmumfang, verbrauchte Zeit oder Fehleranzahl. Die *subjektiven Metriken* erfordern menschliche Einschätzung. Ein Beispiel für eine solche Metrik ist Kundenzufriedenheit. Daten für solche Metriken können Antwortklassen zugeordnet werden, z.B. ausgezeichnet, gut, befriedigend, schlecht. Diese Klassen sollten durch Referenzpunkte auf einer Skala definiert werden.

Absolut/Relativ. *Absolute Metriken* sind invariant gegenüber der Hinzufügung neuer Elemente. Der Programmumfang ist beispielsweise absolut und unabhängig vom Umfang anderer Programme. *Relative Metriken* ändern sich abhängig von anderen Elementen, z.B. der Durchschnitt. Objektive Metriken sind oft absolut, wobei die subjektiven meistens relativ sind.

Direkte/Abgeleitet. *Direkte Metriken*, auch Basis genannt, werden direkt ermittelt, während *abgeleitete Metriken*, auch berechnete Metriken genannt, von anderen direkten oder abgeleiteten Metriken berechnet werden. Ein Beispiel für direkte Metrik ist Ausfallanzahl, abgeleitete – Fehleranzahl geteilt durch LOC.

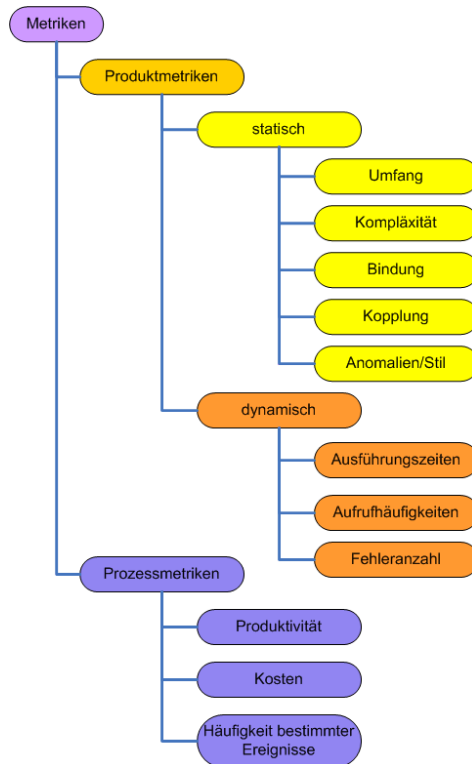


Abbildung 2: Eine Metrik- Taxonomie [Pizka 2004]

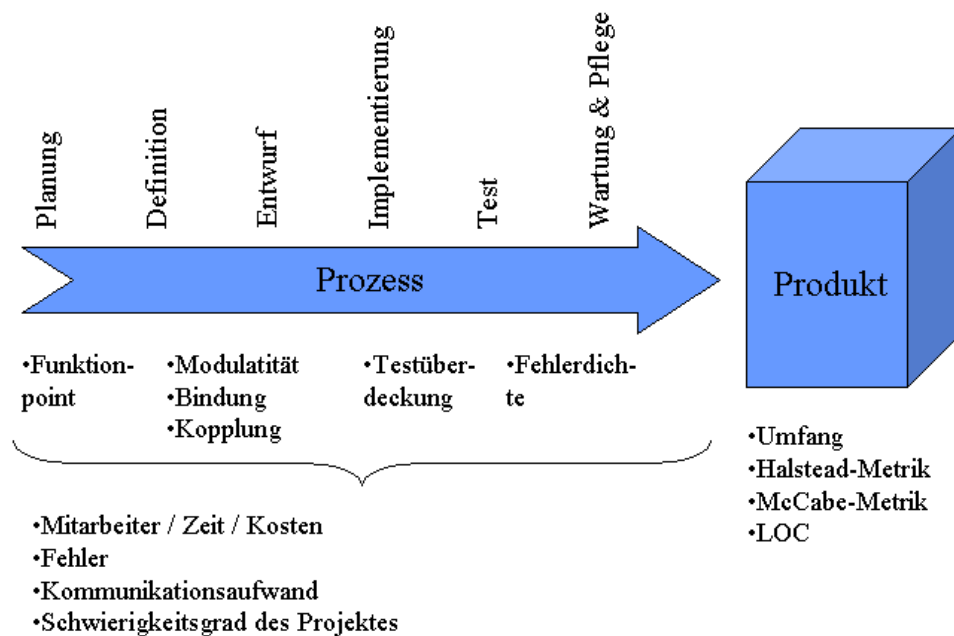


Abbildung 3: Metriken im Software-Entwicklungsprozess [Balzert 1998].

Dynamisch/Statisch. *Dynamische Metriken* besitzen eine Zeitdimension, z.B. gefundene Fehler pro Tag. Die Werte von solchen Metriken ändern sich abhängig davon, wann die Messung durchgeführt wurde. *Statische Metriken* bleiben invariant, z.B. die Anzahl von gefundenen Fehlern in einer ganzen Entwicklung.

Vorhersage/Bewertung. *Vorhersagende Metriken*, oder Abschätzungen können im voraus ermittelt oder generiert werden, während die *Bewertungsmetriken* hinterher ermittelt werden.

Prozessorientiert/Produktorientiert. Eine *Prozessorientierte Metrik* ist ein Attribut des Entwicklungs- und Pflegeprozesses, z.B. Kommunikationsaufwand. Eine *Produktorientierte Metrik* wird am Produkt gemessen. Sie sagt nichts darüber aus, wie das Produkt entstanden ist und warum das Produkt gerade in diesem aktuellen Zustand ist.

Global/Speziell. Die globalen und speziellen Metriken unterscheiden sich in dem Abstraktionsniveau. Die *globalen Metriken* umfassen meistens mehrere Phasen eines Entwicklungsprozesses, z.B. Umfang, Produkt- und Prozessqualität. *Spezielle Metriken* sind Indikatoren für jeweils eine spezielle Phase im Entwicklungsprozess.

3.3 Goal-Question-Metric Paradigma

Viele Software-Metrik Programme scheitern deswegen, weil die Metriken nicht genug *aussagekräftig*, oder sogar ohne definitives Ziel abgeleitet sind. Um den Prozess der Metrik-Definition systematisch zu gestalten, wurden verschiedene Ansätze entwickelt. Nur einige davon sind: „Quality Funktion Deployment Approach“ (QFD) [Balzert 1998], „Software Quality Metrics“ (SQM) [McCall 1977] und „Goal-Question-Metric“ [Basili 1988]. Auf den letzten gehen wir ausführlicher ein.

Goal-Question-Metric Ansatz, kurz GQM ist ein ziel-orientiertes Vorgehen und wurde von Vic Basili und seinen Kollegen von der Universität Maryland entwickelt. Die fundamentale Idee ist einfach: der Prozess für die Definition einer Metrik muss entsprechend folgender drei Schritte gemanagt werden.

1. Setze Ziel entsprechend dem Zweck, Perspektive und der Umgebung
2. Bilde die quantifizierbaren Fragen aus Zielen
3. Um die Fragen zu beantworten, leite die Metriken aus Zielen ab

Abbildung 4 zeigt, wie verschiedene Metriken aus einem Ziel abgeleitet werden können.

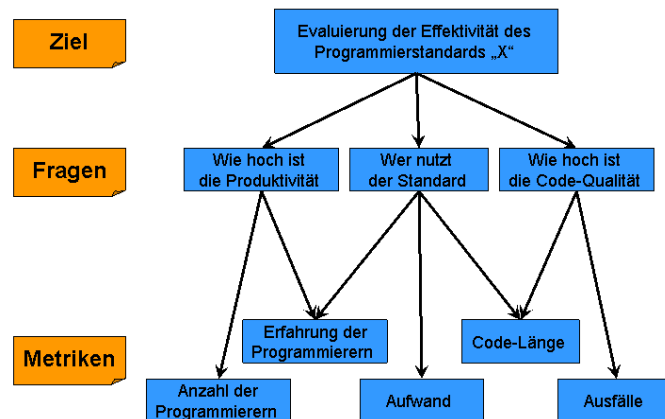


Abbildung 4: Ein Beispiel für die Ableitung von Metriken aus einem Ziel und Fragen.

Das oberste Ziel, der in der Abbildung dargestellten Ableitung, ist die Bewertung der *Effektivität von Nutzung eines Programmierstandards*. Für die Effektivität eines Standards müssen folgende Fragen beantwortet werden.. Als erstes ist es wichtig zu wissen, wer nutzt den Standard. Damit kann die Produktivität von dem, der den Standard nutzt und von dem der den Standard nicht nutzt verglichen werden. Weiter ist es wichtig, die Qualität vom Quellcode mit Standard und ohne zu kennen. Um diese Aspekte zu berücksichtigen, sind die Fragen über Produktivität und Qualität wichtig.

Sobald eine Frage identifiziert ist, muss analysiert werden, was gemessen werden muss, um die Frage zu beantworten. Zum Beispiel, um zu verstehen, wer den Standard nutzt, muss festgestellt werden welcher Anteil von Programmierern benutzt den Standard. Allerdings ist es wichtig, welche Erfahrungsprofile die Programmierer haben, wie lange sie den Standard, die Entwicklungsumgebung, die Programmiersprache und andere Faktoren benutzen, die der Evaluierung beitragen. Die Produktivitäts-Frage basiert auf einer Definition von Produktivität, die oft als die Aufwand-Messung geteilt durch die Messung des Produktumfanges dargestellt wird. Wie in der Abbildung gezeigt, kann dafür eine von folgenden Metriken benutzt werden: LOC, „Function Points“ oder eine andere. Analog dazu kann die Qualität eines Produktes mit Hilfe von gefundenen Fehlern gemessen werden, oder anderen, für einen bestimmten Fall geeigneten Methoden.

Auf diese Weise werden nur diese Metriken bestimmt, die die gewünschten Ziele erfüllen. An der Stelle kann man noch anmerken, dass verschiedene Messungen auch ein einzelnes Ziel erfüllen können, oder mehrere Ziele werden durch eine Messung erfüllt. Das Ziel stellt den Zweck für die Sammlung von Daten auf und die Fragen weisen darauf hin, wie die Daten zu verwenden sind.

3.4 Kritik an Software-Messung

Trotz allen Vorteilen, die uns Metriken in der Software-Engineering bieten, verstecken sich oft in den Software-Messungen einige *schwerwiegende Nachteile*. Die grundsätzliche Problematik von Metriken liegt darin begründet, dass man das, was einen interessiert und man eigentlich messen möchte, nicht direkt messen kann [Balzert 1998]. Beispielsweise will man die Qualität eines Produktes wissen. Um dieses Problem zu lösen, stützt man sich auf Hypothesen. In einer Formel fasst man die quantitative Beziehung zwischen messbaren und interessierenden Größen zusammen. Da jedoch der Software-Entwicklungsprozess noch nicht vollständig verstanden ist, fehlt quantitativen Aussagen eine Basis.

Sehr oft kann man sich leicht in der Definition von *direkten Metriken* täuschen [Kaner 2004]. Nach dem Standard 1061 sind direkte Metriken (für Definition siehe Abschnitt 3.2) unabhängig von allen anderen Attributen, valid und bilden eine Basis für abgeleitete Metriken. Daher ist es mit Vorsicht zu genießen, eine Metrik als direkte zu bezeichnen.

Beispielsweise wird sehr oft Metrik „*Mean Time to Failure*“ (*MTTF*) a priori als direkt definiert. Man kann aber sehr einfach zeigen, dass die Metrik nicht ganz direkt ist. Ihr Wert hängt von vielen anderen Attributen ab [Kaner 2004]. Bei der Fehlererfassung hängt MTTF davon ab, wer arbeitet mit dem zu testenden Programm. Ein erfahrener Spezialist kann ständig die von ihm bekannte Funktionalität benutzen und weniger Ausfälle verursachen als ein Laie, der viele verschiedene Funktionalitäten ausprobieren will. Wenn man die Zeit der Fehlererfassung betrachtet, wird sehr oft auf Tage/Wochen abstrahiert. Das heißt, gleichgültig ob das Programm 10 Minuten oder 8 Stunden pro Tag getestet wird, wird beides als ein Arbeitstag interpretiert. Dazu wird noch oft auf die scharfe Definition von Fehlern verzichtet. Ein Fehler kann ein Programmausfall, korrupte Daten oder eine Fehlermeldung des Programms sein.

Cem Kaner schlägt in seiner Arbeit [Kaner 2004] ein Evaluierungs-Framework für die Bewertung einer Metrik vor. Das Framework basiert auf folgenden 10 Fragen:

1. Was ist der Zweck der Messung?
2. Was ist der Gültigkeitsbereich der Messung?
3. Welche Attribute werden gemessen?
4. Welcher Skala werden die Attribute zugeordnet?
5. Welche Varianz haben die Attribute?
6. Welche Werkzeuge müssen bei der Messung benutzt werden?
7. Welcher Skala wird die Metrik zugeordnet?
8. Welche Varianz können die gemessenen Werte haben?
9. Welcher Zusammenhang besteht zwischen dem Attribut und der Metrik?
10. Welche Nebeneffekte können die benutzten Werkzeuge verursachen?

In der oben genannten Arbeit wird zu dem Framework eine Fallstudie für die Metrik „Fehleranzahl“ durchgeführt. Die Ergebnisse zeigen, dass sogar die einfache Metrik sehr oft falsch interpretiert und als Basis für falsche Aussagen benutzt wird. Zum Beispiel wird die Metrik zur Bewertung der von den Testern gemachten Arbeit benutzt. Allerdings kann man die Leistungen der Tester nicht durch die Anzahl der gefundenen Fehler ausdrücken, obwohl manche Unternehmen auf dieser Basis Prämien auszahlen.

Software-Metriken sind daher mit der notwendigen *Vorsicht und Skepsis* zu betrachten. Sie liefern bestenfalls relative Aussagen und weisen in der Regel auf Anomalien hin, die sowohl positiv als auch negativ sein können.

4 Software-Metriken und Reengineering

Moderne Softwaretechnik kann sich nicht allein mit den Fragen der Entwicklung neuer Softwaresysteme beschäftigen. Der größte Teil der praktisch eingesetzten Software ist bereits mehr als Jahrzehnte alt und bedarf weiterhin einer intensiven Pflege und Wartung [Ebert 2003].

Es gibt, zum Beispiel, einen großen Bestandteil der *Altsoftware* in der Banken-Verwaltung oder in der Fertigungsindustrie, die in alten Programmiersprachen wie Cobol, Fortran oder Assembler implementiert ist. Sie ist auf konventionelle Host-Systeme zugeschnitten, die sich von moderner Systemarchitektur deutlich unterscheiden. Einerseits verursacht die Altsoftware eine Menge von Problemen bei der Wartung, wegen den veralteten Programmiersprachen. Gleichzeitig ist diese Altsoftware aber der Einbindung in die neuen Entwicklungen unterworfen, was eine neue Schnittstelle fordert. Andererseits enthält die Software viel Unternehmenswissen in einer Form von Geschäftsabläufen, die nirgendwo mehr dokumentiert sind. Deswegen stellt sie einen großen Wert dar und ist von besonderer Bedeutung für die Unternehmen.

Reengineering ist ein Oberbegriff für alle Prozesse, deren Ziel eine Überprüfung und Umbau von existierenden Software-Komponenten ist [SRAH 1997]. Es ist sehr eng mit dem Thema Software-Metriken verzahnt und stellt ein gutes Anwendungsbeispiel dafür dar.

In der Arbeit wird das Reengineering am Beispiel eines DoD-Standards „*Software Reengineering Assessment Handbook*“ ([SRAH 1997]) erläutert. Das „*Software Reengineering Assessment Handbook*“ enthält die Angaben und die Anleitung für DoD-Softwaresysteme, insbesondere für Wartung existierender Systeme, Implementierung wiederverwendbarer Komponenten, und Integration „Commercial Off The Shelf“ (COTS)/Non-Developmental Items (NDI) in ein neues oder schon existierendes System.

Das Handbook bietet einen definierten „*Software Reengineering Assessment*“ (SRA) *Prozess* für die Durchführung einer effektiven technischen und ökonomischen Bewertung und Managementaktivitäten für existierende Software, um zu bestimmen, ob die Software zu warten, zu reengineeren, oder einzustellen ist. Der Prozess kann für verschiedene Vorgehensmodelle und verschiedene Typen von Projekten (Eingebettete Systeme, Real-Zeit Systeme oder Waffen-Steuerungssysteme) eingesetzt werden.

Abbildung 5 stellt ein Überblick über SRA vor. Die einzelnen Bestandteile des Prozesses werden detailliert in weiteren Abschnitten beschrieben. Die „*Technische Bewertung*“ (blaue Kasten in der Abbildung 5) identifiziert die Komponenten, die einem Reengineering-Prozess unterliegen und schlägt geeignete für das Reengineering Strategien vor. Im Teil „*Ökonomische Bewertung*“ des SRA-Prozesses (orange Kasten in der Abbildung 5) werden die Strategien für ausgewählte Komponenten ökonomisch bewertet. Die Bewertung basiert auf Kostenabschätzungs-Methoden. Das Management-Prozess (grüne Kasten in der Abbildung 5) unterstützt alle andere Aktivitäten des SRA-Prozesses, sorgt für die Einhaltung von Formalitäten bei Ergebnisauffassung und bringt in die Reengineering-Bewertung unternehmensspezifische Ziele ein.

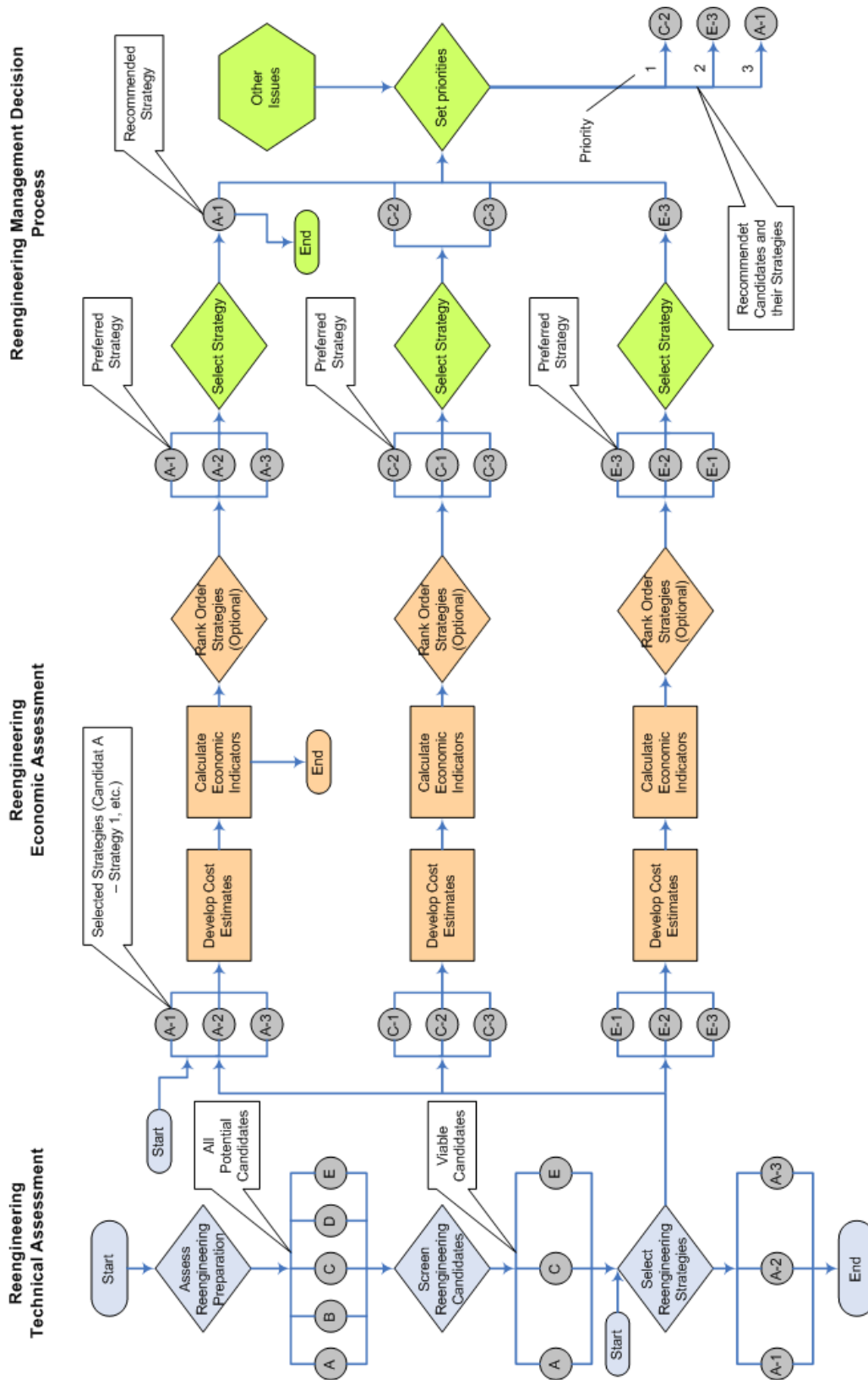


Abbildung 5: Software Reengineering Assesment (SRA) Prozess-Überblick [SRAH 1997]

4.1 Technische Bewertung

Die Technische Bewertung (Reengineering Technical Assessment, kurz RTA) ist der erste Teil des Software Reengineering Bewertungs Prozesses. Das Hauptziel der Bewertung ist die Zuordnung der existierenden Software-Komponenten (z.B. „legacy“ Komponenten) zu einer adäquate Reengineering- oder Wartungs-Strategie. In dem Buch werden insgesamt *acht Strategien* vorgeschlagen. Wobei die ersten sechs sich auf Reengineering beziehen und die letzten zwei die klassischen Maintenance/Support-Strategien sind.

Redocumentation ist ein Prozess für die Analyse eines Systems, mit dem Ziel, eine Dokumentation für Support zu erstellen.

Reverse Engineering ist der Prozess in dem das System verstanden, analysiert und zu einem höheren Abstraktionsniveau abstrahiert wird. Das Abstraktionsniveau ist in dem Kontext des Lebenszyklusses zu verstehen. Zum Beispiel, besteht das klassische Wasserfall-Modell aus folgenden Phasen: Requirements Engineering, Design, Test, Implementierung und Wartung. Wenn wir mit dem Quellcode starten, dann extrahiert das Reverse Engineering aus dem Quellcode ein Design-Modell.

Translate Source Code ist der Prozess der Überführung des Quellcodes von einer Programmiersprache in die andere.

Data Reengineering ist die Transformation von Daten aus einem Format in das andere.

Restructuring ist der Prozess, in dem das existierende System wird aus einer Darstellungsform in die andere auf dem gleichen Abstraktionsniveau überführt. Wobei die „externe Funktionalität“ des Systems beibehalten bleiben muss.

Retargeting ist der Prozess, in dem ein existierendes System auf eine neue Konfiguration transformiert, rehostet oder portiert wird. Zum Beispiel kann die neue Konfiguration eine neue Plattform, ein neues Betriebssystem oder eine „Computer Automated Support Environment“ Plattform (CASE-Plattform) sein.

Redevelopment ist ein Engineering-Prozess für die Entwicklung eines Systems aus einem existierenden konzeptuellen Design, aus Anforderungen oder Spezifikation.

Status Quo - durchgängiger Maintenance/Support für existierende Software.

Das RTA bietet Methoden für die Auswahl der besten aus den acht vorgestellten Strategien für eine gegebene Software-Komponente. In der Abbildung 1 sind die sechs Schritte der technischen Bewertung vorgestellt.

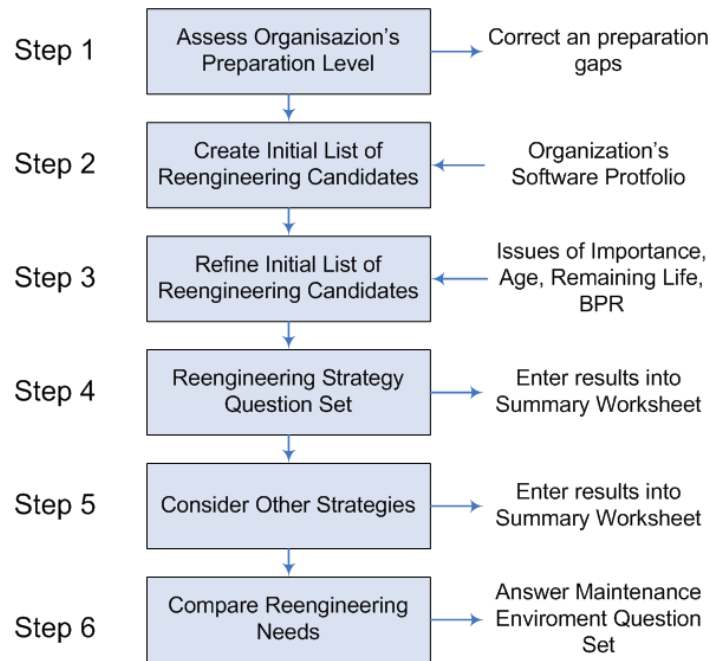


Abbildung 6: Technische Bewertung. Ablauf Diagram [SRAH 1997]

Step One: Assess the Organisation's Level of Preparation. In diesem Schritt wird mit Hilfe von einem Fragenkatalog der Vorbereitungsgrad einer Organisation für Reengineering-Aktivitäten ermittelt.

Step Two: Identify Software Reengineering Candidates. In diesem Schritt werden aus dem Portfolio aller Software Systemen einer Organisation diejenigen Komponenten identifiziert, die ein Problem bereiten, oder ein signifikantes Verbesserungs-Potential haben. Die typischen Kriterien für die Auswahl sind:

- Alter der Komponente,
- Komplexität,
- Programmiersprache,
- Zuverlässigkeit,
- Niveau der Maintenance/Support,
- Dokumentations-Stand,
- die Auswirkung bei Ausfall der Komponente,
- Kopplung zwischen Software und Hardware,
- Persönliche Erfahrungen,
- Randbedingungen für Evolution,
- Randbedingungen bei einem möglichen Plattformwechsel.

Step Three: Reduce the List of Software from Step Two. Entferne aus der Liste die Kandidaten, die einem der folgenden Kriterien genügen:

- Die restliche Lebensdauer der Software-Komponente beträgt nicht mehr als 3 Jahre.
- Der Kandidat ist nicht wichtig genug für Reengineering.

- Der Kandidat wurde erst vor weniger als 5 Jahren entwickelt.
- Der Software-Kandidat unterstützt das zu reengineerende Betriebs-Prozess.

Step Four: Complete the Reengineering Technical Assessment Questions. In dem Schritt werden Fragebogen ausgefüllt entsprechend der fünf Strategien: „Redocumentation“, „Translate Source Code“, „Data Reengineering“, „Retargeting“ und „Restructuring“. Der Fragebogen ist für die Bewertung auf „Retargeting“ als ein Beispiel in der Abbildung 7 vorgestellt.

Retarget	
	<ol style="list-style-type: none"> 1. How portable is the language currently used by the candidate software? <ol style="list-style-type: none"> (1) Language is very portable (2) Portable but significant embedded routines that are not portable (3) Not very portable to other platforms 2. Is the candidate software stand-alone or is it a part of a larger system? <ol style="list-style-type: none"> (1) Candidate software is closely connected to a larger system (2) Candidate software is connected to a larger system but candidate software could be isolated with a well-defined interface (3) No, candidate is a stand-alone 3. What is the age of the current hardware platform? <ol style="list-style-type: none"> (1) Less than 3 years old (2) Three to six years old (3) Over six years old 4. Does the organization need to maintain duplicates of the same candidate software due to different hardware platforms required for the software to execute upon? <ol style="list-style-type: none"> (1) No (2) Not sure, of differences are not significant and easily maintainable (3) Yes 5. Does the new maintenance environment call for CASE tools? <ol style="list-style-type: none"> (1) No (2) Not currently, but will probably move to CASE tools soon (3) Yes 6. Does the current hardware platform cause severe execution problems? <ol style="list-style-type: none"> (1) No (2) Yes, but not frequently (3) Yes 7. Is vendor support for the current hardware guaranteed for remaining software life? <ol style="list-style-type: none"> (1) Yes (2) No, but support loss is unlikely (3) No, and this is an immediate concern 8. Is the current hardware platform limiting expansion or future development activities? <ol style="list-style-type: none"> (1) No (2) To some extent (3) Yes, further software expansion, future development, or new technical advances are limited due to the existing hardware platform

Abbildung 7: Fragebogen für die Bewertung auf „Retargeting“

Step Five: Consider Other Strategies. Ziel dieses Schrittes ist festzustellen, ob eine der drei zusätzlichen Strategien „Reverse Engineering“, „Redevelopment“ und „Status Quo“ auch berücksichtigt werden müssen.

Step Six: Comparing Reengineering Needs Across Different Software Components. Nachdem alle Komponenten einzeln bewertet sind, werden in dem Schritt die Zusammenhänge zwischen einzelnen Strategien anhand von Management-Zielen und Wartungs-Umgebungen evaluiert.

Alle Ergebnisse der RTA-Phase (zu reengineernde Kandidaten mit ihren Strategien) werden festgehalten und als Eingabe für die folgende ökonomische Bewertung benutzt.

4.2 Ökonomische Bewertung

Die ökonomische Bewertung (Reengineering Economic Assessment, kurz REA) ist der zweite Teil des Reengineering-Prozesses. Dieser Teil des Prozesses wird auf den im Teil RTA identifizierten Kandidaten ausgeführt. Das Ziel des REA ist die Entwicklung einer realistischen und konsistenten Kosten-Abschätzung für alle Strategien der Kandidaten. Ein Überblick des Teilprozesses ist in der Abbildung 8 dargestellt.

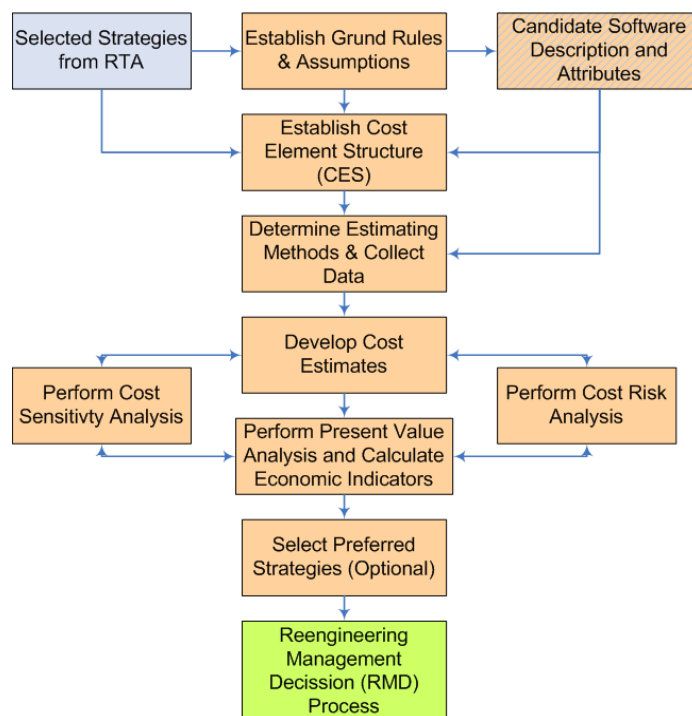


Abbildung 8: Ökonomische Bewertung. Prozess-Überblick [SRAH 1997]

Step one: Establish Ground Rules & Assumptions. Die Randbedingungen und Annahmen (Annahmen über Struktur von Kostenfaktoren, betrachtete Zeit usw.) bieten eine gemeinsame Basis für alle Abschätzungen und sorgen für Management-Flexibilität.

Step two: Establish Cost Element Structure. Die „Cost Element Structure“ (CES) wird für alle Kandidaten und ihre Strategien angewandt. Die Struktur beinhaltet alle Kostenfaktoren, die mit Akquisition, Entwicklung, Wartung, Hardware, Werkzeugen, Tutorials usw. in direkter Verbindung stehen. Der Aufbau der Struktur basiert auf dem „Work Breakdown Structure for Software Elements“, die in dem MIL-HDBK-171 beschrieben ist. Dabei werden nur die Kosten vom Anfang des Reengineering-Aufwandes bis zum Zeitpunkt betrachtet, wann benutzbare Software entwickelt sein wird.

Step three: Determine Estimating Methods & Collect Data. In dem Schritt werden die Abschätzungs-Methoden für jeden Kostenfaktor identifiziert. Die Methoden benutzen die Aufwandsskalen, Preis-Kataloge, Meinungen von Experten usw. Verschiedene Abschätzungs-

Methoden werden typisch für verschiedene Kostenfaktoren eingesetzt. Der Entwicklungs-Aufwand wird normalerweise mit Hilfe von einem parametrischen Modell geschätzt. Er kann aber auch mit Hilfe von historischen Daten oder „best engineering“ Abschätzungen ermittelt werden. Die folgenden parametrischen Modelle werden im REA zur Auswahl vorgeschlagen:

CHECKPOINT ist eine Methode, die auf Kostenvoraussage für Programmierung und sonstige Hauptaktivitäten abgezielt ist. Sie bietet ein komplettes Monitoring des Projektstatus. Die erhobenen Messdaten werden automatisch zu der Projekt-Größe, ausgedrückt in „Function Points“, „Feature Points“ oder LOC, angepasst. CHECKPOINT basiert auf etwa hundert Studien aus industriellen und staatlichen Einrichtungen. Die Messkriterien überdecken nicht nur solche Faktoren wie Personalausbildung, Termine, Kosten, Projektumfang, und Fehleranzahl, sondern auch Personalqualifikation, Umgebungsfaktoren, andere Faktoren und Randbedingungen an der Management- und Kundenseite.

PRICE S (PRICE Software Model) verwendet ein parametrisches Kostenabschätzungsmodell für die Vorhersage von Ressourcen für ein Software-Entwicklungsprojekt. Das Modell basiert auf den programmatischen, qualitativen und quantitativen Eingabeparametern. Unter den *programmatischen Parametern* sind die Termine, die ökonomischen und buchhalterischen Parameter gemeint. Die *qualitativen Parameter* sind: Typ der Programmiersprache, Mitarbeitererfahrung, Werkzeuge, Software-Spezifikationsniveau und Schwierigkeitsgrad der Integration. Die *quantitativen Parameter* bestehen aus einer funktionalen Mischung von Software-, Entwicklungs- und Supportvorgaben, sowie auch dem Umfang der neuen und wiederverwendbaren Software und dem qualitativen Niveau der gelieferten Software.

SEER-SEM (System Evaluation and Estimation of Resources – Software Estimation Model) ist ein Modell, das voraussichtliche Kosten und Terminplanung für solche Projekte wie „Restructuring“, „Retargeting“, „Code Translation“, „Maintenance/Support“, „Incremental Build“ und „neue Entwicklung“ abschätzen kann. Genauer gesagt, SEER-SEM ist ein Werkzeug für die Abschätzung von parametrischen Kosten, Aufwandes, Wartung, Risikos und Terminen. Der Vorteil des Modells besteht in der Benutzung eines sogenannten „Knowledge Bases“. Das „Knowledge Base“ ist eine Sammlung von historischen Daten für solche Software-Entwicklungs-Kategorien wie Plattform, Anwendung, Akquisition, Entwicklungs-Methoden und Standards.

SLIM (Software Life-cycle Management) ist eine Methode für solche Management-Aufgaben wie die Vorhersage des Umfangs, Produktivität, Komplexität und anderen Randbedingungen eines Projektes. Die Vorhersage des Umfangs basiert auf LOC, „Function Points“, Anzahl von Objekten, GUIs, Diagrammen. Die Vorhersage der Produktivität und Komplexität basiert auf solchen Faktoren wie Eigenschaften des Entwicklungsteams, Typ der verwendeten Werkzeuge, Methoden und Anwendungskomplexität. Andere Randbedingungen werden anhand von Terminen, Budget, Zuverlässigkeitsniveau und Information über Mitarbeitern ermittelt.

SOFTCOST-OO ist eine Erweiterung der SOFTCOST-Ada-Methode für die objektorientierte Entwicklung mit C++. Sie enthält folgende fünf Modelle: „Sizing“, „Estimation“, „Allocation“, „Risk“ und „Life cycle“. Die Modelle basieren auf Projektfaktoren (Typ der Software, System-Architektur, Anzahl von involvierten Unternehmen usw.), Produktfaktoren (Grad der Realzeit, Grad der Optimierung usw.), Prozessfaktoren (Niveau der Standardisierung, Gültigkeitsbereich des Supports usw.), Personenfaktoren (Anwendungserfahrung, Entwicklungsumgebungserfahrung usw.) und einigen Vorhersagen bezüglich Aufwandes, Dauern, Konfidenzintervalls usw.

Soweit das akzeptable Modell festgelegt ist, werden alle notwendigen Daten für das Modell gesammelt.

Step four: Develop Cost Estimates. Mit Hilfe von vorgegebenen Templates werden die Kosten für alle Kostenfaktoren abgeschätzt. Optional werden Sensitivity- und Risk-Analyse durchgeführt. Die Sensitivity-Analyse kann für ein parametrisches Modell eingesetzt werden, um die Empfindlichkeit des Modells abhängig von variablen Werten der Parametern zu prüfen. Zum Beispiel, wenn der

Lebenszyklus variiert zwischen 8 und 10 Jahren, um die Empfindlichkeit der Methode zum Lebenszyklus festzustellen, muss die Analyse für beide Werte durchgeführt werden.

Zusätzlich dazu kann in einer Risk-Analyse die Wahrscheinlichkeiten für Risiken und ihre Auswirkungen eingeschätzt werden. DoD's 5000.2-R bietet sieben Faktoren für Risiko-Analyse an: Gedankengang, Technologie, Design, Support, Herstellung, Kosten und Ablaufplan.

Step five: Perform Present Value Analysis and Calculate Economic Indicators. Die Abschätzungen werden summiert, konvertiert in „Present Value“ und benutzt für die Berechnung des „Net Present Value (NPV)“ und „Benefit Investment Ration (BIR)“ für jeden Kandidaten. Laut dem Prozess können auch andere ökonomische Indikatoren wie Break-even Point (BP) und Rate of Return (ROR) benutzt werden, aber im Allgemeinen bereitet es einen zusätzlichen Aufwand.

Das REA benutzt „Total Cost (TC)“ und „Total Benefit (TB)“ für den Vergleich der Kosten und des Gewinnes für jede Strategie. Das „Present Value of the Total Cost (PV(TC))“ und dessen abgeleitete Messungen werden für den Vergleich der Kosten von alternativen Strategien benutzt. Analog dazu wird das „Present Value of Total Benefit“ für den Vergleich der Gewinne von alternativen Strategien benutzt.

Step six: Select Preferred Strategies. In diesem Schritt werden anhand von ökonomischen Indikatoren die Reengineering-Strategien vom Analysten und technischen Personen geordnet. Die Ordnung wird im Weiteren in den Management Entscheidungs-Prozess involviert.

Aus der rein ökonomischen Sicht, ist NPV der meist angemessene Indikator für die „beste“ Strategie. Er widerspiegelt die Strategie, deren Ergebnis den größten netto Gewinn hat. Das BIR kommt auch in betracht als Indikator, um die vorgegebenen Strategien zu ordnen. In Unterschied zu NPV, betont BIR das Verhältnis zwischen dem Gewinn und der Investition.

4.3 Management Prozess

Der Management Entscheidungs-Prozess (Abbildung 9) ist der letzte Schritt im Software Reengineering Assesment (SRA) Prozess. Dieser Abschnitt beschäftigt sich mit den Fragen wie die technische Bewertung und ökonomische Bewertung seine Ergebnisse für Management darlegen müssen und Kriterien für Priorisierung dem Management vorschlagen. Das RMD besteht aus folgenden drei Schritten.

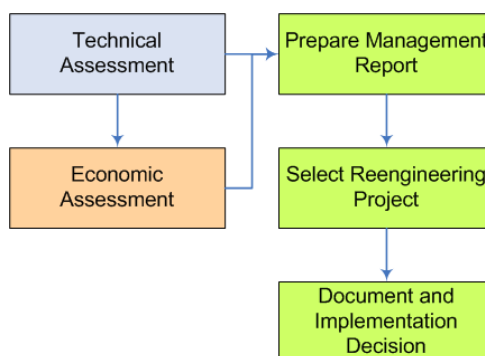


Abbildung 9: Management Entscheidungs-Prozess [SRAH 1997]

Step one: Prepare Management Report. Der Management Report ist notwendiges Dokument in dem Prozess, weil die Person, die die Entscheidung über das Reengineering Projekt trifft, ist nicht in dem REA oder RTA involviert. Die Ergebnisse von diesen Prozess-Teilen müssen formal in einem konsistenten und qualifizierten Format dem Management dargestellt werden. Sogar in dem Fall wenn die oben beschriebene Person doch im REA oder RTA teilnimmt, muss das Dokument als Referenz für

andere Projekte generiert werden. Dieses Dokument soll von Teilnehmern des REA- und RTA-Prozesses geschrieben und für die Auswahl des Reengineering Projektes benutzt werden.

Step two: Reengineering Project Selection. Die Auswahl des Reengineering Projektes besteht aus dem Reengineering Projekt und der Entscheidung über notwendige Ressourcen basierend auf dem Assessment-Ergebnis und Organisations-Faktoren. Die Auswahl ist hauptsächlich die Entscheidung des Management-Personals.

Step three: Implement and Document Decision. Dieser Abschnitt empfiehlt einige Dokumentationen, die für finale Ergebnisse und Implementierung des Reengineering Projektes benutzt werden kann. Diese Dokumentationen und Implementierung sind vorzugsweise vom Management-Personal und dem Reengineering Projekt-Team zu verwenden.

5 Zusammenfassung

Die große Anzahl von Software-Metrik Aktivitäten ist ein positives Zeichen in der Entwicklung der Software-Engineering Disziplin. Metriken dringen in alle Bereiche des Software-Engineering. Wobei an mehreren Stellen die Software-Messung noch immer einige Lücken hat. Zum Beispiel, die Messtheoretische Ansätze werden sehr oft unbeachtet.

Obwohl die Menge der Software-Metriken ständig wächst, bleiben die messtheoretischen Ansätze für alle Metriken gleich. Die Ansätze bilden eine gemeinsame Basis für alle Metriken. Die wissenschaftlich unterstützten Definitionen von Metriken führen zu einem aussagekräftigen Ergebnis in der Messung. Einige Studien zeigen, dass oft in der Praxis mit definierten Metriken falsche Aussagen getroffen werden. Die Metriken haben oft sogar keinen Bezug zu gemessenen Attributen. Nur mit detaillierter, systematischer Analyse der Aufgabe oder des gemessenen Attributes können reproduzierbare und nutzbare Ergebnisse in der Software-Messung erzielt werden.

Trotz vieler Vorteile, die die systematische Einsätze für Definition einer Metrik mit sich bringen, werden sie in der Praxis noch kaum erhoben, was man nicht zuletzt darauf zurückführen kann, dass zumeist noch sehr unausgereifte Entwicklungsprozesse verwendet werden, bei denen Messungen einfach nicht vorgesehen sind und daher der Nutzen auch nicht so klar ist.

Ein Beispiel der systematischen Anwendung von Metriken im Bereich „Reengineering Assessment“ ist mit der Beschreibung des Software „Reengineering Assessment Handbook“ im Kapitel 4 vorgestellt worden. Die Kern-Prozesse der Bewertung sind die Auswahl geeigneter Strategien für Reengineering und die Kostenabschätzung für jede ausgewählte Strategie. Diese Entscheidungen werden benutzt, um die entsprechenden Reengineering-Aktivitäten einzusetzen. Auf diese Weise kann die Effektivität und Effizienz eines Reengineering Projektes gesteigert und unnötige Kosten vermieden werden.

Der Prozess hat eine klare Struktur. Mit Hilfe von Fragebogen und Templates werden die Anwender eindeutig durch den ganzen Prozess durchgeführt. Die Fragen sind meistens so formuliert, dass sie keine Interpretationsfreiheit hinterlassen. Die Abschätzungsmethoden sind ein wichtiger Bestandteil des Prozesses. Sie umfassen ein breites Spektrum von Entwicklungsprojekten: Von den objektorientierten Entwicklungen bis zu den Entwicklungen von Echtzeitsystemen. Diese Methoden bieten eine benutzerfreundliche Werkzeugunterstützung an, damit werden die Anwender von zahlreichen Berechnungen abstrahiert. Neben dem Vorteil, der diese Abstraktion bringt, hat die Beschreibung einen Nachteil: Das Berechnungsweg kann nicht nachvollzogen werden. Der nächste Nachteil resultiert sich daraus, dass der Prozess relativ alt ist. Einige Methoden gehen nur auf die Entwicklung für ganz spezielle Hardwareplattformen ein, die in 90er Jahren verbreitet waren. Aktuelle Hardwareplattformen werden von diesen Methoden nicht unterstützt.

Abkürzungen und Begriffe

CASE	Computer Automated Support Environment
CES	Cost Element Structure
Data Reengineering	Data Reengineering ist die Transformation von Daten aus einem Format ins andere.
DoD	Department of Defense
Forward Engineering	Forward Engineering ist eine Menge von Aktivitäten, die auf einem Basis von neuen Anforderungen und mit Produkten und Artefakten von existierenden Systemen ein neues System erstellen. Das Unterscheid zwischen Software-Engineering und Forward Engineering ist in dem, dass das Forward Engineering basiert sich auf anderen Reengineering-Aktivitäten. Zum Beispiel, das Forward Engineering kann ein Nachfolger von Reverse Engineering sein.
GQM	Goal Question Metric Paradigm
LOC	Lines of Code
MTTF	Mean Time to Failure
REA	Reengineering Economic Assessment
Redevelopment	Redevelopment ist ein Engineering-Prozess für Entwicklung eines Systems aus einem existierenden konzeptuellen Design, Anforderungen und Spezifikation.
Redocumentation	Redocumentation ist ein Prozess für die Analyse eines Systems, mit dem Ziel, eine Dokumentation für Support zu erstellen.
Reengineering	Reengineering ist die Überprüfung und Umbau von existierenden Software-Komponenten. Dieses Prozess beinhaltet eine Kombination von Subprozessen: Reverse Engineering, Restrukturierung, Translating, Data Reengineering, Redocumentation, Forward Engineering und Retargeting. Der Schwerpunkt liegt in der Konsolidierung, Migration oder Wiederverwendung.
Restructuring	Restrukturierung ist der Prozess, in dem das existierende System wird aus einer Darstellungsform in die andere auf dem gleichen Abstraktionsniveau überführt. Wobei die externe Funktionalität des Systems muss beibehalten bleiben.

Retargeting	Retargeting ist der Prozess in dem ein existierendes System wird transformiert, rehostet oder portiert auf eine neue Konfiguration. Zum Beispiel die neue Konfiguration kann eine neue Plattform, neue Betriebssystem oder eine CASE-Plattform sein.
Retirement	Einstellung einer Software-Komponente.
Reverse Engineering	Reverse Engineering ist der Prozess in dem das System wird verstanden, analysiert und abstrahiert zu einem höheren Abstraktionsniveau. Das Abstraktionsniveau ist in dem Kontext des Lebenszyklusseees zu verstehen. Zum Beispiel, das klassische Wasserfall-Modell besteht aus folgenden Phasen: Requirements-Engineering, Design, Code, Test, Implementierung und Wartung. Wenn wir starten mit Code, dann extrahiert das Reverse Engineering davon ein Design-Modell.
RTA	Reengineering Technical Assessment
Status Quo	Durchgängige Maintenance/Support für existierende Software.
Maintenance/Support	Der Prozess in dem die existierende Software wird geändert, wobei die Hauptfunktionalität wird beibehalten. Software

Tabellenverzeichnis

Tabelle 1: Übersicht über verschiedene Maßskalen.

9

Abbildungsverzeichnis

Abbildung 1: Berechnung der McCabe-Metrik	5
Abbildung 2: Eine Metrik- Taxonomie [Pizka 2004]	10
Abbildung 3: Metriken im Software-Entwicklungsprozess [Balzert 1998].	10
Abbildung 4: Ein Beispiel für die Ableitung von Metriken aus einem Ziel und Fragen.	12
Abbildung 5: Software Reengineering Assesment (SRA) Prozess-Überblick [SRAH 1997]	15
Abbildung 6: Technische Bewertung. Ablauf Diagram [SRAH 1997]	17
Abbildung 7: Fragebogen für die Bewertung auf „Retargeting“	18
Abbildung 8: Ökonomische Bewertung. Prozess-Überblick [SRAH 1997]	19
Abbildung 9: Management Entscheidungs-Prozess [SRAH 1997]	21

Literaturverzeichnis

- [Albrecht 1979] Albrecht, A.J.: Measuring Applications Development Productivity. Proceedings of IBM Applic. Dev. Joint SHARE/GUIDE Symposium, Monterey, CA, 1979, pp.83-92.
- [Balzert 1998] Balzert, H.: Lehrbuch der Software-Technik, Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung; Spektrum, Berlin, 1998.
- [Basili 1988] Basili V.R., Rombach H.D.: The TAME project: Towards improvement-oriented software environments, IEEE Transactions on Software Engineering 14(6), pp 758-773, 1988.
- [Belady 1979] Belady, L.A. On Software Complexity In: Workshop on Quantitative Software Models for Reliability. IEEE No. TH0067-9, New York, N.Y., pp.90-94, Oktober 1979.
- [Boehm 1981] Boehm, B.W.: Software Engineering Economics. Prentice Hall, 1981.
- [Chidamber 1994] Chidamber S.R, Kemerer C.F A Metrics Suite for Object Oriented Design, in IEEE Transactions on Software Engineering, 20(6), 1994, S. 476-493.
- [Ebert 2003] Jürgen Ebert. Software-Reengineering - Umgang mit Software-Altlasten. Konradin-Verlag. Grasbrunn. Informatiktage 2003. Seiten 224-31.
- [Fenton 1991] Fenton, N.: Software Metrics: A Rigorous Approach, Chapman & Hall, 1991.
- [Fenton 1994] Fenton, N.: Software Measurement: A Necessary Scientific Basis, in: IEEE Transactions on Software Engineering, März 1994, S. 199-206
- [Finkelstein 1984] Finkelstein, L.: A review of the fundamental concepts of measurement, Measurement, vol. 2, no. 1, pp. 25-34, 1984.
- [Grady 1992] Grady, Robert B.: Practical Software Metrics for Project Management and Process Improvement. Prentice Hall 1992.
- [Halstead 1977] Halstead, M.H.: Elements of Software Science. New York, Elsevier North-Holland, 1977.
- [Harrison 1981] Harrison, Warren; Magel, Kenneth: A Complexity Measure Based on Nesting Level. ACM SIGPLAN Notices, Volume 16, No. 3, pp. 63-74, 1981.

- [Hecht 1977] Hecht, M.S: Flow Analysis of Computer Programs. Elsevier, New York, 1977.
- [Helmer-Heidelberg 1966] Helmer-Heidelberg, O: Social Technology, Basic Books, New York, 1966.
- [Kaner 2004] Kaner C., Bond W.P.: Software Engineering Metrics: What Do They Measure and How Do We Know? 10th International Software Metrics Symposium, Metrics 2004.
- [Lorenz 1994] Lorenz M., Kidd J.: Object-oriented Software Metrics, Prentice Hall, 1994.
- [McCabe 1976] McCabe, T.: A Complexity Measure. IEEE Transactions of Software Engineering, Volume SE-2, No. 4, pp. 308-320, Dezember 1976.
- [McCall 1977] McCall J.A., Richards P.K., Walters G.F.: Factors in Software Quality, RADC TR-77-369, 1977. Vols I,II,III', US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055, 1977.
- [McClure 1978] McClure, Carma L.: A Model for Program Complexity Analysis. 3rd International Conference on Software Engineering, Mai 1978, pp. 149-157.
- [Morris 1989] Morris, Kenneth, L.: Metrics for Object-Oriented Software Development Environments. Massachusetts Institute of Technology, Master of Science in Management, Mai 1989.
- [Nelson 1966] Nelson, E.A. : Management Handbook for the Estimation of Computer Programming Costs. AD-A648750, Systems Development Corp., 1966.
- [Laemmel 1977] Laemmel, A.; Shooman, M.: Statistical (Natural) Language Theory and Computer Program Complexity. POLY/EE/E0-76-020, Dept. of Electrical Engineering and Electrophysics, Polytechnic Institute of New York, Brooklyn, August 15, 1977.
- [Oviedo 1980] Oviedo, Enrique I.: Control Flow, Data Flow and Programmers Complexity. Proceedings of COMPSAC 80, Chicago IL, pp.146-152, 1980.
- [Park 1992] Park, Robert, E.: Software Size Measurement: A Framework for Counting Source Statements. Software Engineering Institute, Pittsburg, SEI-92-TR-020, 220 pages, Mai 1992.
- [Pizka 2004] Pizka, M.: Unterlagen zur Vorlesung Software Metriken; Vorlesungsreihe: Vertiefung ausgewählter Themen des Software Engineering, TUM, 2004.

- [Roberts 1979] Roberts, F.S.: Measurement Theory with Applications to Decision Making, Utility, and the Social Sciences. Reading, MA: Addison Wesley, 1979.
- [Rubey 1968] Rubey, R.J.; Hartwick, R.D.: Quantitative Measurement Program Quality. ACM, National Computer Conference pp. 671-677, 1968
- [Ruston 1981] Ruston, H.: Software Modelling Studies: The Polynomial Measure of Complexity. RADC-TR-81-183, Rome Air Development Center, Air Force Systems Command, Griffis Air Force Base, Rome, NY, July 1981.
- [Shepperd 1993] Shepperd, Martin (Editor): Software Engineering Metrics - Volume I: Measures and Validations. McGraw Hill Book Company, International Series in Software Engineering, 1993.
- [SRAH 1997] Software Reengineering Assessment Handbook. Technical Report, (JLC-HDBK-SRAH) Version 3.0, 1997
- [Troy 1981] Troy, Douglas; Zweben, Stuart: Measuring the Quality of Structured Design The Journal of System and Software. Volume 2, 113-120, 1981, pp.113-120.
- [Walston 1977] Walston, C. E.; Felix, C.P.: A Method of Programming Measurement and Estimation. IBM Systems Journal, Vol. 16, No. 1, pp. 54-73, 1977. Also in: Tutorial on Programming Productivity: Issues for the Eighties, IEEE Computer Society, Second Edition, 1986.
- [Wolverton 1974] Wolverton, R.W.: The Cost of Developing Large-Scale Software. IEEE Transactions on Computer, Volume C-23, No. 6, pp. 615-636, June 1974. Also in: Tutorial on Programming Productivity: Issues for the Eighties, IEEE Computer Society, Second Edition, 1986.
- [Zuse 1998] H. Zuse: History of Software Measurement; <http://irb.cs.tu-berlin.de/~zuse/metrics/3-hist.html>, 1998.