FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Barrieren des semi-automatisierten Architectural Recovery

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Michael Hablich
Matrikelnummer 0200045

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek
Mitwirkung: -

Wien, 20.10.2011 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Michael Hablich, Pelargonienweg 23/21, 1220 Wien

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 10.11.2011

Danksagung

Folgenden Personen möchte ich explizit danken:

Meine Freundin Marion hat mich während der stressigen Diplomarbeitszeit maßgeblich physisch und psychisch unterstützt.

Meine Eltern Sonja und Arnulf haben es mir ermöglicht zu studieren.

Mein Betreuer, Professor Futschek, der immer eine Anlaufstelle für methodische Fragen war und mich erinnert hat, sich auf das Wesentliche zu konzentrieren.

Meine Haustiere Jojo, Leni, Chio und Sonic, weil sie einfach da sind.

Kurzfassung

Architectural Recovery (AR) bezeichnet die Methode des Reverse Engineering (RE) von Software-Architektur. In dieser Diplomarbeit wird das semi-automatisierte Architectural Recovery genauer betrachtet. Dies sind Methoden, welche durch menschlichen Input befähigt sind, eine Architektur zu beschreiben oder den User bei der Beschreibung zu unterstützen.

Es soll im Rahmen dieser Arbeit herausgefunden werden, wie es um die derzeitige Einsatzfähigkeit des semi-automatisiertes Architectural Recovery steht. Es wurden mehrere bekannte Techniken identifiziert, um ein solches Reverse Engineering durchzuführen. Eine Evaluation der aktuellen Implementierungen dieser Ansätze ermöglichte das Auffinden von aktuellen Barrieren. Die Bewertung ist anhand von QSOS erfolgt.

Die Grenzen beim Einsatz von semi-automatisierten Architectural Recovery umfassen vier grundsätzliche Kategorien. Organisatorische Rahmenbedingungen der eingesetzten Tools können einen großen Einfluss auf ihre Einsatzfähigkeit haben. Zum Beispiel kann eine nicht vorhandene Dokumentation der Methoden-Implementierung eine wesentliche Barriere bei ihrer Anwendung sein. Des Weiteren sind die Usability und die visuelle Präsentation der gesammelten Daten eine Herausforderung für die Tools. Bei der Code-Analyse wird zwischen dynamischer und statischer Analyse unterschieden. Vor allem erstere bieten eine große Herausforderung für das semi-automatisierte Architectural Recovery.

Es kann gesagt werden, dass viele der aufgezeigten Barrieren lösbar sind, wenn die nötigen Ressourcen vorhanden sind.

Abstract

Architectural Recovery (AR) is the method of reverse engineering of software architecture. The focus of this thesis is the semi-automated Architectural Recovery. This is a collection of methods aimed at assisting humans with their Architectural Recovery efforts.

The intent of this thesis is the detection of current barriers in the field of semi-automated Architectural Recovery to measure its utilisability. To get a grasp of the current state several well know techniques were identified. An evaluation of implementations of these approaches allowed the identification of current barriers. QSOS was used as the evaluation framework.

The limitations on the use of semi-automated Architectural Recovery comprise four basic categories. The organizational environments of the tools can have a major impact on its operational capability. For example a non-existing documentation of an implementation can be a major risk in using the application. Furthermore, the usability and the visual presentation of the collected data is a challenge for the tools. Code analysis compromises of two types: static and dynamic analysis. The latter offers a great challenge for the semi-automated Architectural Recovery.

Finally, it can be said that many of the barriers identified are solvable, if the necessary resources are available.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Bei der Erstellung von Software fallen in der Regel mehr Artefakte an, als an einen Standard-Kunden ausgeliefert werden. Normalerweise erhält dieser das ausführbare Programm und die Programmdokumentation. Bei größeren Projekten ist es aber oft der Fall, dass der Kunde das Produkt noch weiterentwickeln und warten will. Probleme ergeben sich dabei, wenn nur der Source-Code ohne ausreichende Dokumentation für die Weiterentwicklung vorhanden ist. Durch das Fehlen von Artefakten, die die Struktur und das Verhalten der Software im Detail beschreiben, erschwert sich die Einarbeit in das Projekt erheblich. Ist der Source-Code auch nicht dokumentiert, gibt es als Einarbeitungspunkte nur mehr das Pflichtenheft (bzw. den Vertrag) und die Programmdokumentation.

Um diese Problemstellung zu lösen, kann man Reverse Engineering einsetzen. Das Ziel dieses Vorgangs ist es, genug Informationen über das Produkt zu sammeln, damit eine den Anforderungen genügende Software-Dokumentation vorliegt - sei dies nun als Use-Case-Diagramm, Klassendiagramm oder Source-Code-Kommentar. Der Sinn ist, die Weiterentwicklung stark zu vereinfachen.

Im Rahmen dieser Diplomarbeit werden semi-automatisierte Methoden und Tools des Reverse Engineerings angewendet und auf ihre Nützlichkeit beim Architectural Recovery untersucht. Dadurch soll herausgefunden werden, was die derzeitigen Barrieren im Architectural Recovery sind.

1.2 Forschungsfragen

Folgende Fragen sollten in der Diplomarbeit beantwortet werden:

- In wieweit ist es möglich, Architectural Recovery semi-automatisiert auszuführen?
- Welche Barrieren treten dabei auf?
- Sind dies prinzipielle Limitierungen oder wurde in diese Richtung nur noch nichts entwickelt?
- Können auch dynamische Zusammenhänge analysiert werden?

- Welche Unterschiede in Bezug auf die Barrieren gibt es bei dynamischen und statischen Zusammenhängen?

1.2.1 Vermutete Ergebnisse

Folgende Ergebnisse werden vorab vermutet:

- Grenzen bei der statischen Analyse sind technisch gesehen nicht vorhanden.
- Barrieren in der Implementierung können Dependency Injection Frameworks wie Spring sein.
- Eine Analyse zur Laufzeit gestaltet sich um einiges komplexer als das statische Pendant. Zusätzlich ist das Durchführen einer dynamischen Analyse stark technologieabhängig.

2 Reverse Engineering

Da Architectural Recovery ein Teil des Reverse Engineerings ist, wird dieses zuerst beschrieben.

2.1 Definition von Reverse Engineering (RE)

Reverse Engineering bezeichnet den Vorgang, den Aufbau und die inneren Abläufe eines Systems herauszufinden. Die dabei gewonnen Erkenntnisse sollen in einer höheren Abstraktionsstufe präsentiert werden [ChCr90]. Forward Engineering bezeichnet den klassischen Softwareentwicklungsprozess.

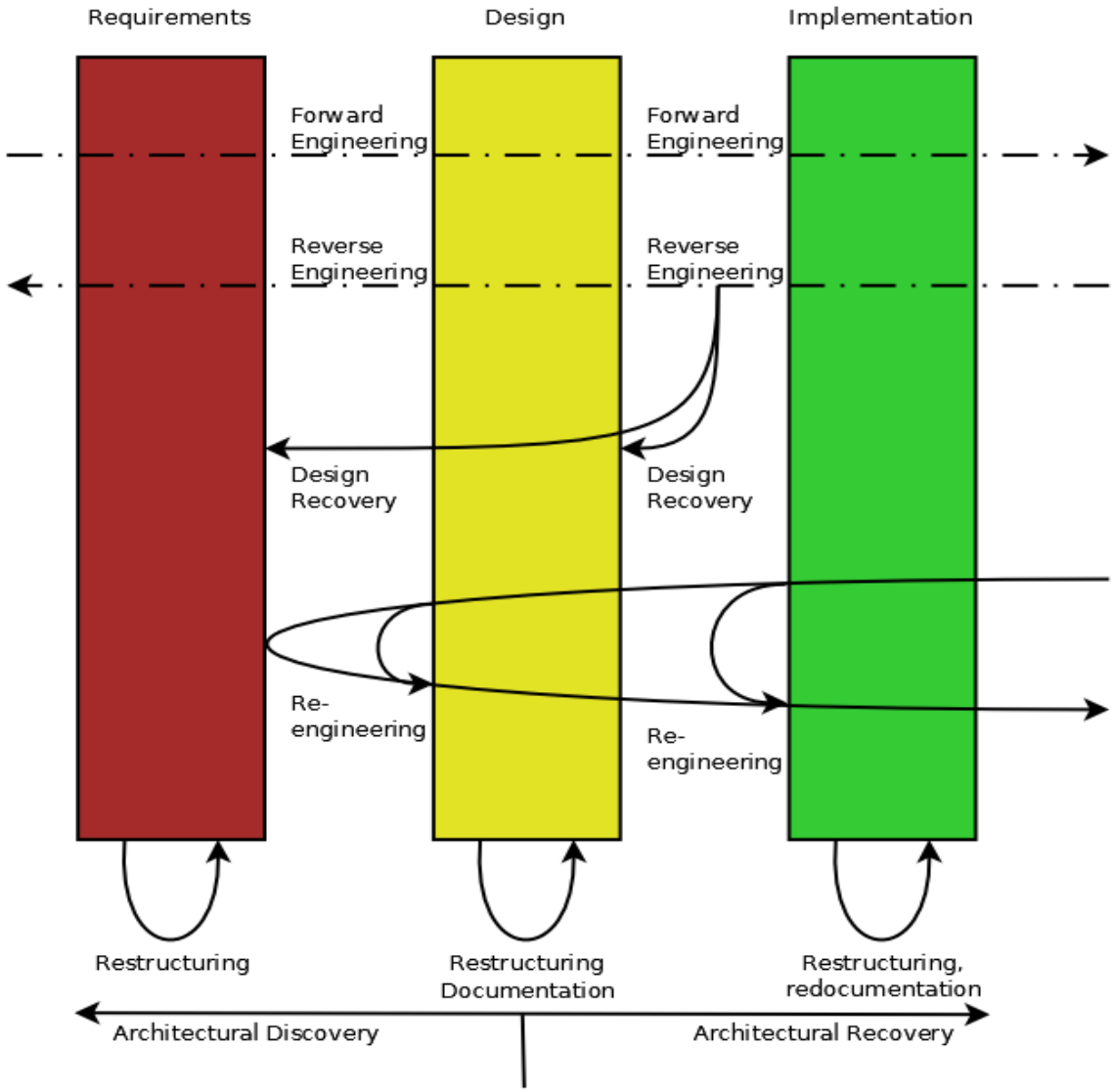

Abbildung 1: Zusammenhänge und Beziehungen von Forward Engineering und Reverse Engineering

Auf Basis einer abstrakten Systembeschreibung wird das System implementiert

[ChCr90]. Folgende Spezialisierungen von Reverse Engineering existieren laut Chikofsky und Cross [ChCr90]:

- *Redocumentation* erzeugt eine andere Sicht auf dem gleichen Abstraktionsniveau
- *Design Recovery* erzeugt eine Sicht auf das System von einem höheren Abstraktionsniveau aus gesehen.

Artefakte einer Software können in der Regel in drei Gruppen aufgeteilt werden:

- Source-Code (Implementierungsartefakte)
- Design-Artefakte
- Anforderungen

Diese drei Teile sind miteinander verbunden und bauen aufeinander auf.

2.2 Methoden des RE

Diese Kapitel befasst sich mit den Arten, wie Reverse Engineering durchgeführt werden kann.

2.2.1 Beobachtende, nicht invasive Analyse

Eine beobachtende, nicht-invasive Analyse betrachtet das System als Black-Box und nimmt keine Veränderung im Programm vor. Grundsätzlich sind damit folgende Methoden gemeint:

- Dokumentation der Verhaltensweise des Systems
- Analyse der Datendateien
- Differenzanalyse

Dabei ist zu beachten, dass diese Ansätze das Verhalten eines normalen Anwenders imitieren.

2.2.2 Tracing

Beim Tracing werden die Interaktionen der Anwendung mit dem Hostsystem

mitgeschnitten. Dies kann zum Beispiel der Zugriff auf andere Bibliotheken oder die verwendeten Datei-Handles sein. Programme wie Process Explorer[1] ermöglichen solche Analysen. Zusätzlich wird der verwendete Speicher analysiert. Der Unterschied zum Debugging ist, dass beim Tracing statische Daten erhoben werden.

2.2.3 Debugging

Debugging ist eine dynamische Untersuchungsmethode. Damit es möglich, die Instruktionen Schritt für Schritt durchzugehen. In der Regel unterstützen Debugger auch die Möglichkeit, den aktuellen Ausführungscode zu verändern und abzuspeichern.

2.2.4 Disassemblierung

Erfolgt eine Disassemblierung einer Anwendung, wird der vorhandene Objekt-Code in Assembler-Befehle übersetzt.

2.2.5 Dekompilierung

Das Ziel einer Dekompilierung ist es, den Objekt-Code in Quelltext der Ursprungssprache des Programmes zu übersetzen. Laut Cifuentes [Cifu94] gibt es folgende sequentiell abgearbeitete Phasen während einer Dekompilierung:

1. Syntax analyzer
2. Semantic analyzer
3. Intermediate code generator
4. Control flow graph generator
5. Data flow analyzer
6. Control flow analyzer
7. Code generator

Vereinfacht gesagt übersetzt ein Compiler den Source-Code in Maschinenbefehle (oder eine Zwischensprache, die zur Laufzeit interpretiert wird). Aus diesen Kompilaten kann ebenfalls Code reverse-engineered werden. Zu beachten ist, dass bei der Kompilierung

1 http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx

oft Optimierungen ausgeführt und Meta-Daten (wie Kommentare) entfernt werden. Es kann also nicht garantiert werden, dass der generierte Quellcode dem Original entspricht.

2.2.6 Architectural Recovery (AR)

Architectural Recovery ist das Darstellen der Software-Architektur von bereits bestehenden Programmen. Dabei umfasst diese Aufgabe den Prozess einer Herstellung von Architekturinformationen aus der Implementierung [MeEgGr03]. Mehr Informationen zu diesem Thema gibt es im Kapitel Architectural Recovery im Detail.

2.2.7 Architectural Discovery

Beschreibt den Prozess der Architekturaufklärung mit Hilfe der Anforderungen [MeEgGr03]. In einem Reverse Engineering Projekt ist dies Teil der Tasks.

3 Architectural Recovery im Detail

3.1 Software-Architektur

Basierend auf dem *IEEE Standard 1471-2000* [IEEE00] ist Software-Architektur folgendes:

> *„The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.“*
> *[IEEE00], Seite 3*

Die Beschreibung einer Software-Architektur sollte laut dieser Definition den strukturellen Aufbau der Komponenten enthalten. Weiters müssen ihre Beziehungen innerhalb und nach außen beschrieben sein. Es soll auch aufgezeichnet werden, warum etwas genau so gelöst wurde. Dies wird noch genauer im Kapitel Minimale Anforderungen an eine Architekturbeschreibung erläutert.

3.2 AR-Methoden

In diesem Kapitel werden einige verbreitete Methoden ausgeführt, die für das AR verwendet werden.

3.2.1 Programm-Analyse

Der Prozess der Programm-Analyse beinhaltet Methoden, welche automatisiert Informationen über das Programmverhalten zur Laufzeit sammeln. Dabei wird bei der statischen Programm-Analyse das System nicht ausgeführt. Die Analyse erfolgt also auf Basis von Source- oder Objekt-Code. Die dynamische Analyse erwirbt die Daten zur Laufzeit des Programms.

3.2.1.1 Statische Analyse

Dabei haben statische Analysen zwei wesentliche Charakteristika [WCCWWM95]: Das Ziel der Programm-Analyse (*nature*) und die Tiefe (*depth*). Letztere gibt an, wie detailliert die Analyse sein soll. Eine sehr große Tiefe ist das Durchführen eines formalen Beweises, dass der Algorithmus korrekt ist. Im Gegensatz dazu ist die Auflistung der Packages eine sehr kleine Tiefe.

Effektiv werden statische Methoden angewandt, um durch Approximation ein Wissen über das Laufzeitverhalten zu haben [NNH04].

3.2.1.2 Dynamische Analyse

Wie bereits beschrieben, erfolgt die Analyse des Programms während der Laufzeit. Dies ermöglicht unter anderem die Sammlung folgender Daten:

- Crash-Dumps
- Code-Abdeckung
- Speicherverbrauch
- Thread-Analyse
- Nachrichtenwege

Es ist natürlich zu beachten, dass dynamische Analysen das Programmverhalten beeinflussen. Da ein zusätzlicher Code ausgeführt wird, verändert sich die Performance des Systems automatisch. Weiters kann der eingeschleuste Code natürlich auch Fehler enthalten und dadurch das Host-System beeinträchtigen.

3.2.2 Software-Clustering

Clustering beschreibt den Prozess, Objekte mit ähnlichen Eigenschaften in eine Gruppe zusammenzufassen. Software-Teile, die viele Verbindungen zueinander haben, gehören auch zusammen und bilden Cluster. Hier kann es zu Problemen im Bereich von Utility-Komponenten kommen[PiAlHa09]. Da diese Software-Module keine aussagekräftigen Information über die Architektur des Systems beinhalten, sollten sie bei einem AR-Prozess nicht berücksichtigt werden. Utility-Komponenten haben bei der statischen wie auch bei der Laufzeit-Analyse einen großen Einfluss. Beispielsweise würden generierte Sequenzdiagramme Calls auf die Utility-Komponenten beinhalten. Diese Information ist für den AR-Prozess aber eigentlich wertlos.

Um solche Utility-Komponenten zu erkennen gibt es verschiedene Möglichkeiten [PiAlHa09]:

- Fan-In-Analyse

- Impact-Analyse
- Namenskonventionen
- Utility-Kategorisierung

Jedes Tool welches eine Utility-Erkennung unterstützt muss dementsprechend flexibel sein, da eine Definition einer Utility-Komponente von Person zu Person beziehungsweise von Projekt zu Projekt unterschiedlich sein kann.

3.2.3 Design Pattern Detection (DPD)

Design Patterns (Entwurfsmuster) wurden von den Gang of Four (kurz GoF) im Buch „Design Patterns: Elements of Reusable Object-Oriented Software“[GHJV94] aufgezeichnet. Entwurfsmuster dienen als generische Lösung für immer wieder auftretende Probleme in der Software-Entwicklung. Dabei wurden die Design Patterns in drei Kategorien aufgeteilt:

- *Creational patterns* dienen zur Erzeugung und korrekten Initialisierung von Objekten
- *Structural patterns* ermöglichen es, größere Strukturen aus vielen Einzelteilen zusammenzusetzen.
- *Behavioural patterns* organisieren die Kommunikation und Beziehungen zwischen Objekten.

Wie von Maggioni [Magg09] beschrieben, können Design Patterns nicht nur beim Forward-Engineering, sondern auch beim Reverse Engineering nützlich sein. Das Vorhandensein von Entwurfsmustern kann Hinweise darauf geben, warum die Architektur genau diese Ausprägung hat. Natürlich kann das Vorhandensein von Entwurfsmustern auch auf eine gute Architektur-Qualität Hinweise liefern, da Entwurfsmuster generische Lösungen sind. Durch ihre Existenz wird, in der Regel, die Architektur wiederverwendbarer.

3.2.3.1 Software-Micro-Structures

> *"With the term software micro-structure (micro-structure for brevity) we*

indicate any code element that can be automatically and univocally detected from the source code of a software system, and which represents useful basic hints for the understanding of the structures composing a system." [Magg09], Seite 12

Entsprechend dieser Definition enthalten Mikro-Strukturen also auch Informationen über das Verhalten und die Struktur des Systems. Im Besonderen sind diese Informationen zu Beginn eines AR-Projekts interessant, um ein initiales Verständnis für das Systems zu bekommen.

3.2.4 Design Flaws

Design Flaws und Code Smells können eingesetzt werden um die Architektur zu begreifen. Diese Anti-Patterns können ebenso wie Design-Patterns beim Reverse Engineering verwendet werden.

Beispielsweise kann der Design-Flaw „Data Class“ [LaRa06] Hinweise darauf geben, welche Datenschnittstelle verwendet wird. Dies wird vor allem dann relevant, wenn es um versteckte Datenschnittstellen im System geht. Angenommen, das System soll umkonfiguriert werden: Die Hälfte der eingestellten neuen Parameter werden aber nicht vom System angenommen. Eine Suche nach dem Design Flaw „Data Class“ bringt zu Tage, dass es noch eine versteckte XML-Schnittstelle gibt, welche das System konfiguriert.

3.2.5 Code-Metriken

Code-Metriken werden in der Regel zur Sicherstellung der Code-Qualität verwendet. Manche Code-Metriken lassen sich aber im Sinne des AR verwenden.

3.2.5.1 Coupling

Diese Code-Metrik bezeichnet die Abhängigkeit einer Komponente von einer Zweiten. Dabei gibt es verschiedene Stufen des Couplings.

Abhängigkeiten zwischen zwei mutmaßlichen Komponenten weisen darauf hin, dass es sich eigentlich dabei um eine Komponente handelt. Diese Information kann etwa verwendet werden, wenn man zu Beginn des AR-Prozess ein Clustering mittels

Packages/Namespaces gemacht hat. Wenn zwei Namespaces eine gegenseitige Abhängigkeit haben, können diese in einem Cluster zusammengefasst werden.

3.2.5.1.1 Afferent Coupling

Die Anzahl der eingehenden Verbindungen bestimmt den Grad der Verantwortung/Wichtigkeit einer Komponente. Eine große Anzahl an eingehenden Verbindungen weißt darauf hin, dass dieser Cluster wichtig für das System ist.

3.2.5.1.2 Efferent Coupling

Durch das Efferent Coupling wird ausgedrückt, wie viele Verbindungen nach außen gehen. Dadurch wird die Abhängigkeit von anderen Komponenten ausgedrückt. Eine typische Komponente mit hohem Efferent Coupling wäre zum Beispiel eine Plugin für das zu untersuchende Software-System.

3.2.5.2 Cohesion

Kohäsion beschreibt den Spezialisierungsgrad einer Komponente. Typische Utility-Komponenten werden eine niedrige Kohäsion haben, da diese verschiedene Funktionen wahrnehmen. Im AR kann man über diese Metrik herausfinden, wo sich Utility-Komponenten befinden, um sie in der Architektur-Beschreibung extra herauszuheben.

Im Software-Engineering will in der Regel eine möglichst hohe Kohäsion erreicht werden. Dadurch fällt es leichter den Code zu verstehen und zu warten, da dieser nur eine Aufgabe ausführt.

3.2.5.3 Code-Coverage

Der Test-Abdeckungs-Grad bezeichnet den Prozentsatz, den die Tests im Bezug auf den zu testenden Code abdecken. Es gibt verschiedene Kriterien, einen Deckungs-Grad zu berechnen.

Diese Metrik ist interessant für das AR, wenn die Tests dazu herangezogen werden, die Architektur zu verstehen. Vor allem Unit-Tests liefern sehr viele Informationen, da dadurch der erwartete Programmfluss aufgezeigt wird. Weiters kann eine Komponente mit einer hohen Coverage wichtiger sein als ein Programmteil mit niedriger

Testabdeckung.

3.2.5.4 Abstractness

Der Abstrahierungs-Grad setzt die Menge der abstrakten Klassen und Interfaces den Implementierungen innerhalb der Komponente entgegen. Komponenten mit hohem Abstrahierungs-Grad sind besonders interessant, da sie meist als Basis für andere Komponenten dienen.

3.2.5.5 Overview Pyramid

Michele Lanza und Radu Marinescu haben die Overview-Pyramid entworfen, welche für das Reverse Engineering interessante Metriken übersichtlich aufzeigt[LaRa06].

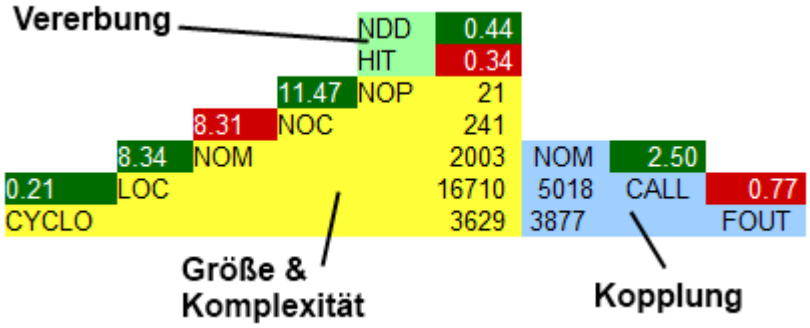

Abbildung 2: Overview-Pyramide von XStream

In Abbildung 2 ist erkennbar, dass die Pyramide in drei Bereich aufgeteilt ist. Diese kennzeichnen die Kategorien, in welche die Metriken eingeteilt sind. Die Zahlen in der Mitte sind die jeweils errechneten Werte. So hätte das Beispiel aus Abbildung 2 16710 Zeilen Code (LOC). Folgende Metriken werden dabei verwendet:

- *CALL* Anzahl der einzigartigen Funktions-/Methoden-Aufrufe
- *CYCLO* Zyklomatische Komplexität
- *FOUT* Fan out
- *HIT* Durchschnitt der Höhe des Vererbungsbaumes
- *LOC* Anzahl der Code-Zeilen
- *NDD* Durchschnitt der Anzahl der Nachkommen
- *NOC* Anzahl der Klassen

- *NOM* Anzahl der Methoden
- *NOP* Anzahl der Packages

Im Inneren der Pyramide werden die absoluten Werte dargestellt. Die Komma-Zahlen an den Wänden der Pyramide werden aus dem unteren und oberen Wert errechnet. Beispielsweise wird 0.21 über CYCLO und LOC ausgerechnet, wie in Abbildung 3 gesehen werden kann.

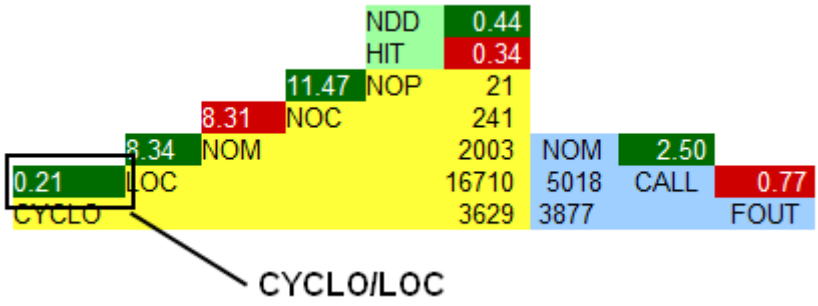

Abbildung 3: Errechnung der Relations-Werte in der Overview Pyramid

Für diese Zahlen gibt es gewisse Bereiche, in welche sie fallen sollten. Ansonsten ist dies ein Hinweis für ein Design-Problem.

3.2.6 Concern-Graphen

Die Methode der Concern-Graphen wurde von Martin P. Robillard und Gail C. Murphy entwickelt [RoMu02]. Dabei werden Code-Stücke sogenannten Concerns - also ihrem Aufgabengebiet - zugewiesen. Dies ist vergleichbar mit Aspekten in der aspektorientierten Programmierung.

Vor allem im Bereich der Software-Wartung kann diese Methode sehr gut eingesetzt werden. Angenommen, es soll das Logging eines bereits bestehenden Legacy-Systems erweitert werden: Der mit dieser Aufgabe anvertraute Software-Engineer kennt die Architektur des Systems nicht. Mittels initialer Nachforschungen definiert er den Concern „Logging“. Auf Basis dieses Concerns werden die restlichen Code-Strukturen erkannt, die das Logging betreffen.

3.2.7 Software Reflexion Models

In „*Software Reflexion Models: Bridging the Gap between Source and High-Level Models*“ [MuNoSu95] wird das Konzept der *Software Reflexion Models* vorgestellt. Diese Modelle zeigen den Unterschied zwischen einer gedachten und einer vorhandenen Architektur. Dabei wird zuerst das aktuelle System von dem Reverse-Engineer beschrieben. Dies ist das *High-Level Model* der untersuchten Architektur. Informationen über den Aufbau des Models, bekommt er aus der aktuellen Dokumentation oder Interviews. Als nächster Schritt wird ein *Source Model* aus dem bestehenden Quellcode generiert. Diese beiden Modelle werden mittels eines deklarativen Mappings miteinander verbunden. Das dadurch generierte *Software Reflexion Model* zeigt auf, an welchen Stellen sich das *High-Level* Model und das *Source Model* voneinander unterscheiden.

Mittels des *Software Reflexion Models* kann aufgedeckt werden, an welchen Stellen sich die gedachte oder überlieferte Architektur von der tatsächlichen unterscheidet.

3.2.8 Software-Evolution

Innerhalb der Einsatzzeit eines Software-Systems kommt es in der Regel zu einigen Änderungen im System. Dies kann man als Evolution bezeichnen, da Anpassungen die Software *fitter* für den User machen. Die Überlebensfähigkeit der Software steigt also.

Im Artikel „*Evolution in software systems: foundations of the SPE classification scheme*“ [CHLW06] werden drei Typen von System definiert:

- *S-type* sind Programme, die formal spezifiziert werden können.
- *P-type* sind Progamme, die nicht formal spezifiziert werden können. Deswegen muss ein iterativer Prozess zur Lösungsfindung verwendet werden.
- *E-type* sind Programme, die in die reale Welt eingebettet sind. Dadurch werden sie auch Teil dieser und verändern sie. Daraus entsteht eine Feedback-Schleife, in der das Programm und die Umgebung sich gegenseitig weiterentwickeln.

In „*Metrics and laws of software evolution-the nineties view*“ [LRWPT97] werden acht Gesetze aufgelistet, nach denen sich ein *E-type* Software-System weiterentwickelt.

1. *Continuing Change:* Das System muss sich immerwährend weiterentwickeln und anpassen. Sollte dies nicht der Fall sein, werden mit der Zeit die Anforderungen nicht mehr erfüllt.
2. *Increasing Complexity:* Wenn sich ein *E-type*-System weiterentwickelt, wird es gezwungenermaßen immer komplexer. Diesem kann man mit Reduktion der Features und genereller Wartung (Refactoring) entgegenwirken.
3. *Self Regulation:* Das System und die sich anpassenden Prozesse regulieren sich gegenseitig.
4. *Conservation of Organisational Stability*: Im *E-type*-System ist die effektive, durchschnittliche, globale, eingesetzte Arbeitszeit über die gesamte Lebensspanne unveränderlich.
5. *Conservation of Familiarity:* Stakeholders des Systems sollten immer mit dem System vertraut sein. Da dieses aber stetig wächst, fällt diese Aufgabe immer schwerer. Deshalb kann Wachstum nur logarithmisch erfolgen.
6. *Continuing Growth:* Der funktionelle Umfang der Software muss sich über ihre Lebensspanne immer mehr erweitern, damit User-Anforderungen befriedigt werden können.
7. *Declining Quality:* Die Software-Qualität eines Systems nimmt mit der Zeit ab, außer es werden explizite Schritte dagegen unternommen.
8. *Feedback System:E-Type*-Systeme sind Teil einer Feedback-Schleife. Deshalb müssen sie während ihrer Lebenszeit auch als solcher behandelt werden. Dies bedeutet, dass das System Einfluss auf seine Umgebung hat.

Natürlich geht es bei dem Thema Software-Evolution auch um andere Themen als die pure Klassifizierung der zu beschreibenden Architektur. Beispielsweise ist es wichtig zu wissen, welcher Entwickler wann eine Änderung durchgeführt hat. Dadurch kann der Ansprechpartner sehr gut für eine nicht verstandene Komponente gefunden werden.

3.3 Visualisierungsarten

Dieses Kapitel beschreibt bekannte Arten der Software-Visualisierung.

3.3.1 Dependency-Diagramme

Dependency-Diagramme sind graphische Abbildungen von Abhängigkeits-Graphen [HoRe92]. Ein Abhängigkeits-Graph ist ein gerichteter Graph, der die Beziehung eines Programmteils zu einem anderen beschreibt. Wenn es eine Verbindung von der Komponente A zu B gibt, dann ist A von B abhängig.

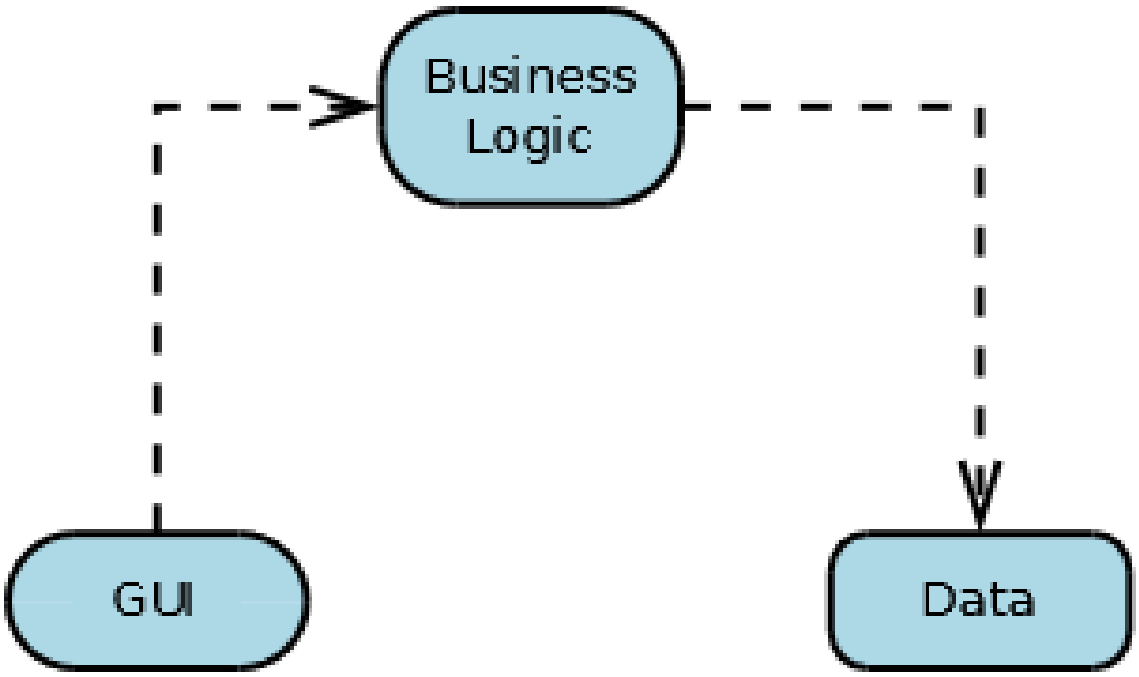

Abbildung 4: Dependency-Diagramm

In Abbildung 4 kann ein Beispiel für ein Dependency-Diagramm betrachtet werden. Die Komponente GUI ist von Business Logic abhängig. Diese wiederum von der Komponente Data.

3.3.2 Tree-Maps

Die Visualisierung Tree-Map wurde von Ben Shneiderman das erste Mal publiziert [Shne91]. Das erste Einsatzgebiet war die Visualisierung der Verwendung von Festplattenspeicher.

Eine Tree-Map zeigt hierarchische Daten an, die als Baum strukturiert sind. Die Äste eines Knotens werden als Teilfläche des Knotens dargestellt. In Verbindung mit einer Kolorierung kann so auf einen Blick eine Kategorierisierung der Daten vorgenommen werden.

Angenommen, die Daten sind nach ihren Packages und Klassen strukturiert. Die Farbgebung erfolgt anhand der Häufigkeit des Aufrufs von "*System.out.println(String)*" (Java-Code für die Ausgabe an den Standard-Output) in den einzelnen Klassen. Je roter die Farbe des Rechtecks, desto mehr höher die Anzahl der Aufrufe. Während des AR-Prozesses wurde herausgefunden, dass log4net verwendet wird. Ein Aufruf von "*System.out.println(String)*" sollte nicht mehr nötig sein und kein Rechteck sollte rot sein. Wenn doch rote Flecken vorhanden sind, kann dies auf ein eventuelles zweites, selbstgebautes Logging-Framework hinweisen. Dies ist natürlich ein sehr vereinfachtes Beispiel.

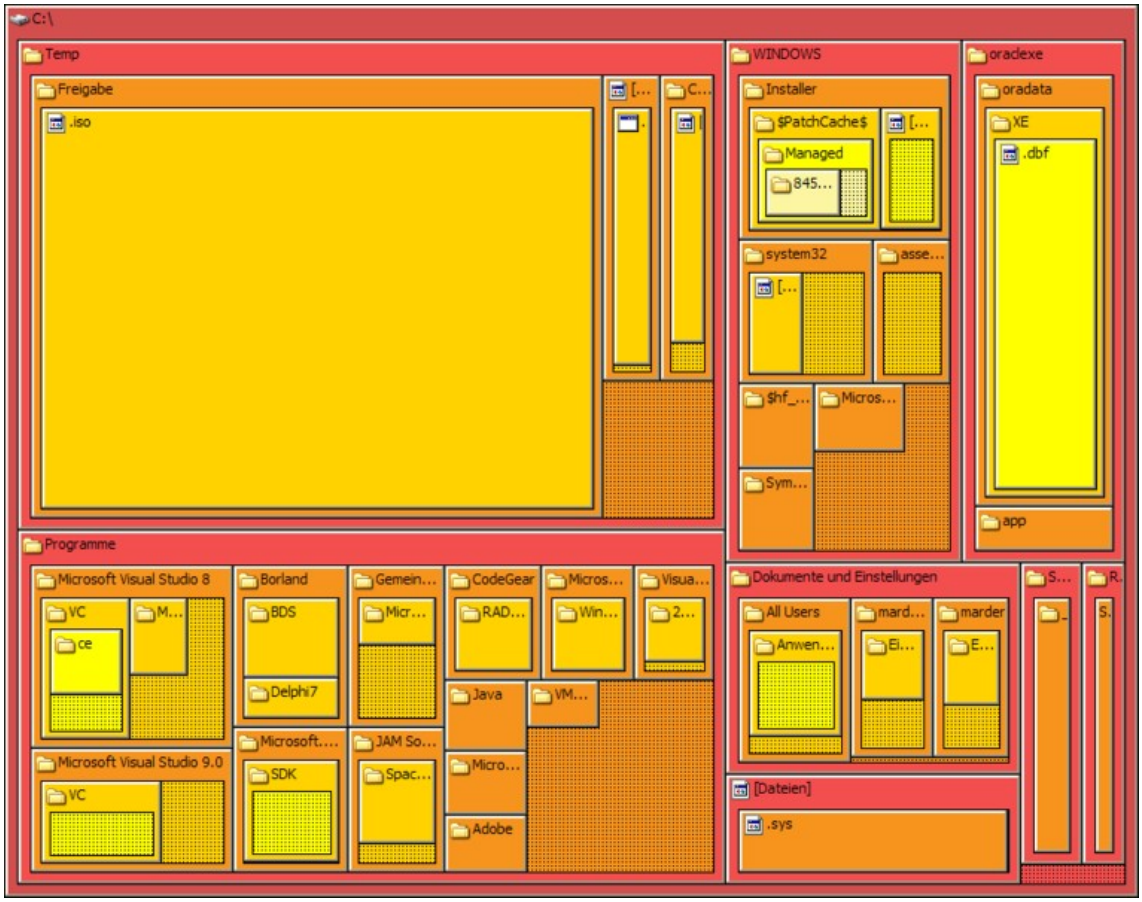

Abbildung 5: Tree-Map die den Inhalt eines Laufwerks visualisiert

Abbildung 5 macht den Inhalt eines Laufwerks für das Auge besser erfassbar. In diesem Fall beschreibt die Farbe das Nesting-Level der Datei – je roter desto höher in der Hierarchie. Die Größe der Rechtecke gibt an, wie groß der Inhalt der Datei oder des Verzeichnisses ist.

3.3.3 Polymetric Views

Polymetric Views [LaDu03] sind zweidimensionale Diagramme die neben Farben, geometrische Eigenschaften der Diagramm-Nodes als Darstellung von Metriken verwenden. Dadurch können bis zu 5 Metriken parallel angezeigt werden: Die verwendeten Eigenschaften sind folgende:

- Farbe des Nodes
- Position des Nodes
- Länge des Nodes
- Breite des Nodes
- Die Verbindungen zu einem anderen Node

Weitere Metriken können durch das Hinzufügen zusätzlicher Merkmale natürlich auch angezeigt werden. So ist es möglich, die Färbung eines Knotens zu schraffieren, wenn es sich um eine Klasse handelt, die in Java realisiert ist. Eine voll ausgefüllte Färbung würde eine C++-Realisierung darstellen.

3.3.4 UML und seine Bedeutung im AR

Häufig sind UML-Diagramme die Ziel-Artefakte eines AR-Projekts. Aber nicht jedes Artefakt bringt in einem vernünftigen Aufwand zusätzliche Information. In diesem Kapitel soll erläutert werden, für welches Aufgabengebiet gewisse Diagramme nützlich sind.

3.3.4.1 Klassendiagramm

Dieses Diagramm ist das am öftesten verwendete UML-Diagramm [DoPa06] in SE-Projekten.

Aus einem Klassendiagramm lässt sich sehr leicht Source-Code generieren, der bereits die Struktur des Programms vorgibt. Umgekehrt ist dies natürlich auch möglich und bietet dadurch zusätzlich noch die Möglichkeit Round-Trip-Engineering zu betreiben.

3.3.4.2 Komponentendiagramm

Komponentendiagramme erfüllen einen ähnlichen Zweck wie Klassendiagramme [JRHZQ07]. Sie visualisieren die Zusammenhänge zwischen den einzelnen Komponenten der Architektur.

3.3.4.3 Use-Case-Diagramm

Use-Case-Diagramme sind vor allem dann interessant, wenn es darum geht, eine initiale Verständnisses des Systems herzustellen. Sie bieten die Möglichkeit, eine Einsicht in gewollte Abläufe zu erhalten.

3.3.4.4 Aktivitäts- und Sequenz-Diagramm

Aktivitäts- oder Sequenz-Diagramme bieten die Möglichkeit Abläufe innerhalb der Software darzustellen. Beim Reverse Engineering ist es deswegen sinnvoll, gezielt Ablaufdiagramme generieren zu lassen.

Viele CASE-Tools sind in der Lage mittels statischer Code-Analyse Sequenz-Diagramme zu generieren. Dependency-Injection-Frameworks bieten hier eine große Herausforderung. Eine statische Code-Analyse kann nur schwer erkennen, welche Implementierer eines Interfaces wirklich verwendet wird. Eine Analyse zur Laufzeit gibt mehr Informationen Preis.

3.4 Definition eines erfolgreichen AR-Projekts

Die IEEE hat einen Standard [IEEE00] definiert, welcher vorschlägt, wie Software-Architektur beschrieben werden soll. Im Dokument ist von „*architecture of a software-intensive system*“ die Rede. In diesem Dokument wird ein Meta-Model für die Architekturbeschreibung vorgestellt. Der Grundgedanke ist dabei immer, dass Architektur die Wünsche der Stakeholders berücksichtigen soll. Stakeholders haben verschiedene Interessen bzw. Hintergründe. Deshalb hat jede Interessengruppe eine andere Sicht auf das System und dessen Architektur. Es wird empfohlen die Beziehungen der Begriffe im Standard [IEEE00] nachzusehen, da dieser ein übersichtliches Klassendiagramm diesbezüglich enthält.

Eine Architektur-Beschreibung sollte laut Standard folgende Punkte enthalten:

- Die Architektur-Beschreibung soll mittels einer ID identifizierbar sein. Zusätzlich soll es einen kurzen Überblick geben.
- Es müssen die Stakeholder und ihre Concerns aufgezeigt werden, sollten diese relevant für die Architektur sein.
- Die gewählten Viewpoints müssen spezifiziert werden. Natürlich muss auch erklärt werden, warum dieser Viewpoint zur Archiektur-Darstellung verwendet wird.
- Einen oder mehrere Views
- Inkonsistenzen innerhalb der Architektur-Erläuterung sollen aufgezeichnet werden.
- Der Grund warum genau diese Architektur gewählt wurde sollte ebenfalls aufgeschrieben werden.

3.4.1 Minimale Anforderungen an eine Architekturbeschreibung

Laut *IEEE Standard 1471-2000* [IEEE00] sollen minimal folgende Punkte einer Architektur beschrieben sein. Zu beachten ist dabei, dass diese Empfehlungen für SE-Projekte gedacht sind, die noch eine Software erstellen müssen. Als Stakeholders ist es empfehlenswert wenn Users, Aquirers (Käufer), Developers und Maintainers identifiziert werden. Da dies in der Regel die wichtigsten Interessengruppen in einem SE-Projekt sind, ergibt diese Empfehlung Sinn.

An Concerns sollten die folgenden festgehalten werden:

- Das Ziel des Software-Systems
- Den Grad, wie gut die Software das Ziel erreicht hat
- Die Wahrscheinlichkeit dieses Systems umzusetzen
- Die Risiken, die während des SE-Projekts anfallen
- Wartbarkeit, Erweiterbarkeit und Einsatzfähigkeit der Software

Dies sind natürlich Punkte, die jedes erfolgreiche SE-Projekt analysieren sollte. Bei einem AR-Projekt kann man den gleichen Ansatz wählen.

Viewpoints müssen immer im Zusammenhang mit Stakeholder und Concerns gebracht werden. Kann dies nicht geschehen, so sind noch nicht alle Concerns und Stakeholders identifiziert. Dies kann als Überprüfung dafür dienen, ob die Analyse der System-Umgebung abgeschlossen ist. Wenn ein Viewpoint zur Architektur-Beschreibung hinzugefügt wurde, muss es dafür einen Grund (Rationale) geben. Ist keiner vorhanden, so kann auch kein Mehrwert aus diesem zusätzlichen Viewpoint abgeleitet werden. Dies wäre also eine klassische Resourcenverschwendung.

Views sind die Ausprägungen der Viewpoints. Deswegen darf sich jeder View immer nur auf genau einen Viewpoint beziehen – ansonsten kommt es zu einer Vermischung von verschiedenen Sichtweisen auf das System, worunter die Qualität der Architektur-Darstellung leidet.

Wie bereits erwähnt muss auch die Konsistenz innerhalb des Systems analysiert werden. Treten Inkonsistenzen auf, so müssen diese unbedingt dokumentiert werden.

3.4.2 Object-Oriented Reengineering Patterns

Im Buch „Object-Oriented Reengineering Patterns“ [DeDuNi08] wird ein Vorgehensmodell für Reengineering-Aufgaben vorgestellt. Da vor einem Reengineering in der Regel ein Reverse Engineering stattfindet, kann dies als Leitfaden für ein erfolgreiches AR-Projekt dienen.

Grundsätzlich werden im Buch vier Hauptteile während des Reverse Engineerings beschrieben. Zu diesen sind jeweils so genannte Patterns zugeteilt. Diese Vorlagen sind die Tasks die in einem Schritt ausgeführt werden.

3.4.2.1 Phase 1: Richtung setzen (Setting Direction)

Zu Beginn eines Re-/Reverse Engineering-Projekts muss wie in jedem Projekt der Grundstein gelegt werden. Die einzelnen Patterns erinnern sehr stark an typische Aktivitäten in einem Forward-Engineering-Projekt. Ein Reverse Engineering-Projekt sollte sich auch an die Best-Practices eines IT-Projekts halten. Es gibt eine gemeinsame Definition der Ziele (*Agree on Maxims*). Diese Ziele sollen innerhalb der Gruppe propagiert werden (*Speak to the Round Table*). Zusätzlich soll sicher gestellt werden, dass das Projekt in die richtige Richtung im Sinne der Projektziele läuft (*Appoint a Navigator*).

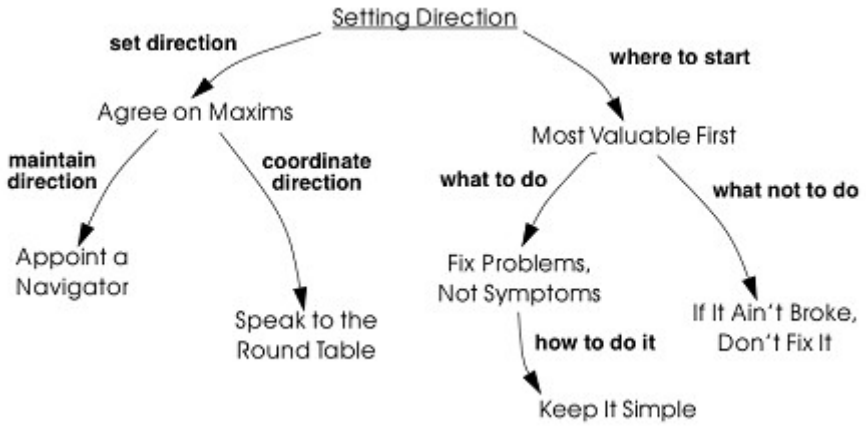

Abbildung 6: Aufgaben in Schritt 1

Quelle: [DeDuNi08] unter der Lizenz http://creativecommons.org/licenses/by-sa/3.0/

Der Rechte Branch befaßt sich mit den Aufgaben in einem Reengineering-Projekt. Eine Priorisierung der Arbeitspakete muss vorgenommen werden (*Most Valuable First*). Da in der Regel die benötigten Resourcen argumentiert werden müssen, wird nur dies reengineered, was auch fehlerhaft beziehungsweise im Sinne der Projektziele ist. Der Grundgedanke jeder Bug-Fixing-Aufgabe oder eines Feature-Einbaus (Reengineering ist in der Regel nichts anderes), ist es einen vorhandenen Fehler zu beheben. Dies wird auch so in der vorhandenen Fachliteratur für Entwickler empfohlen [McCo04]. Das KISS-Prinzip[2] [HuTh99] soll natürlich ebenfalls angewandt werden (*Keep It Simple*).

2Keep It Simple and Straightforward

3.4.2.2 Phase 2: Erster Kontakt (First Contact)

Im Schritt First Contact wird das erste Mal das System untersucht. Der Prozess ist iterativ und in zwei Zonen geteilt. Im oberen Kästchen in Abbildung 7 werden Informationen von bestehenden Entwicklern (*Chat with the Maintainers*) und anderen Stakeholder (*Interview During Demo*) gesammelt.

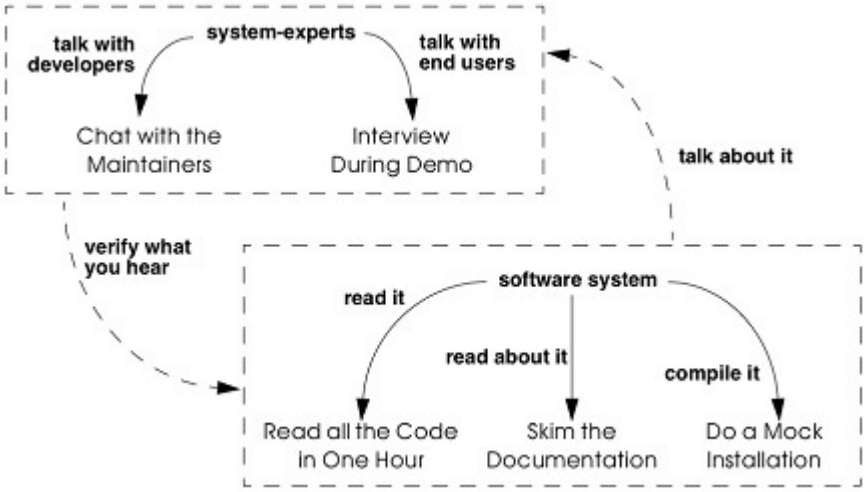

Abbildung 7: Aufgaben in Schritt 2

Quelle: [DeDuNi08] unter der Lizenz http://creativecommons.org/licenses/by-sa/3.0/

Die Daten werden im zweiten Hauptschritt verifiziert. Dies wird durch drei Tasks erreicht. Der Code selbst soll überprüft werden (*Read all the Code In One Hour*). Die Restriktion mit einer Stunde soll sicher stellen, dass das System mit hoher Aufmerksamkeit und Fokus gelesen wird. Die User-Dokumentation ist auch eine Quelle um Informationen zu verifizieren (*Skim the Documentation*). Sollten Diskrepanzen zwischen dem bestehenden System und der User-Dokumentation festgestellt werden, muss die Qualität der Interviews oder der Dokumentation in Frage gestellt werden. Für die späteren Reengineering-Maßnahmen muss das Ziel-System kompiliert werden können. Eine Überprüfung, ob alle nötigen Artefakte und Informationen dafür vorhanden sind, erfolgt ebenfalls (*Do a Mock Installation*).

3.4.2.3 Phase 3: Erste Erkenntnisse (Initial Understanding)

In diesem Schritt werden erste Teile des Systems verstanden. Dies soll der Grundbaustein für die weiteren Reverse-/ und Reengineering-Maßnahmen des Projekts

sein. Die Informationen aus Phase 2 sollen verwendet werden, die Ziele aus Phase 1 auf Machbarkeit zu überprüfen.

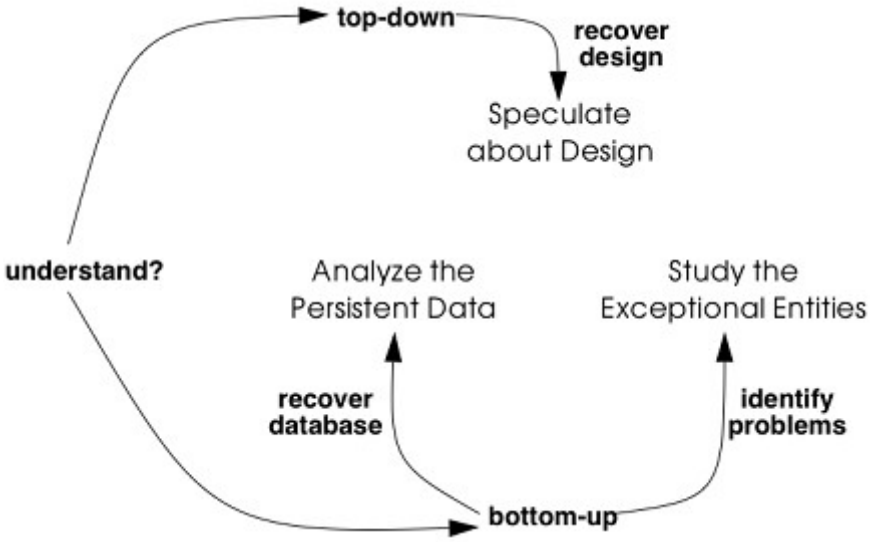

Abbildung 8: Aufgaben in Schritt 3

Quelle: [DeDuNi08] unter der Lizenz http://creativecommons.org/licenses/by-sa/3.0/

Eine bestimmte Reihenfolge, in der die einzelnen Schritte ausgeführt werden, ist nicht vorgegeben. Des weiteren sind auch diese Schritte iterativ anzuwenden.

Aufgeteilt ist diese Phase in *top-down* und *bottum-up*. Dies sind die Richtungen in welche die Architektur wiederhergestellt wird. Der einzige *top-down*-Schritt befaßt sich mit der Rekonstruktion des Designs. Dabei wird die vermutete Architektur für eine Problemstellung durch einen erfahrenden Entwickler ausgedacht und mit dem vorhandenen Source-Code verglichen (*Speculate about Design*). Das wird solange iterativ ausgeführt, bis eine zufriedenstellende strukturelle Beschreibung (meist in Form eines Klassendiagramms) vorhanden ist.

Daten, die in einer externen Datenbasis gespeichert werden, sind in der Regel wertvoll. Leider entspricht dies oft nicht der Wahrheit. Es sollen also die wertvollen Daten in der Datenbasis erkannt und beschrieben werden *(Analyze the Persistent Data*).

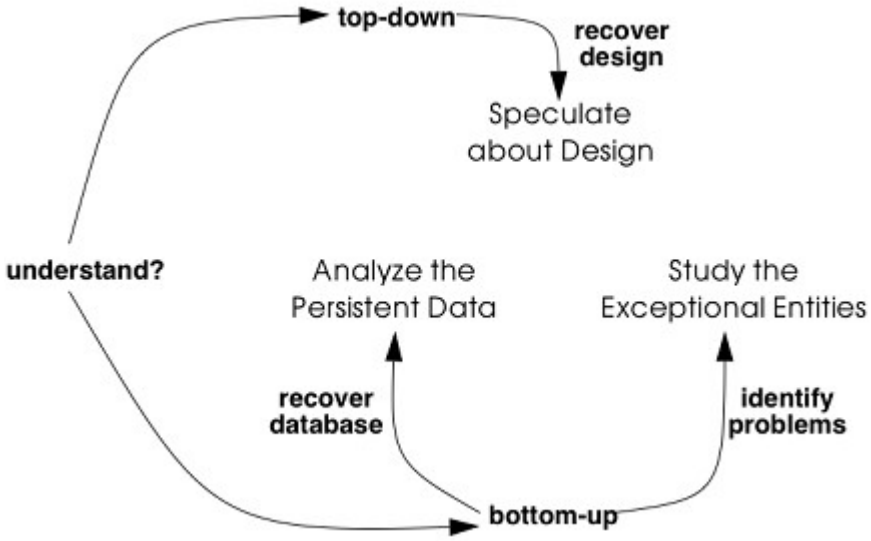

Abbildung 9: Aufgaben in Schritt 3

Quelle: [DeDuNi08] unter der Lizenz http://creativecommons.org/licenses/by-sa/3.0/

Um potentielle Fehler im Design festzustellen, ist es sinnvoll, außergewöhnliche Metrik-Werte im System zu finden (*Study the Exceptional Entities*). Sollten Ausreißer auftauchen, muss die betreffende Stelle genauer im Source-Code überprüft werden. Es gilt zu klären, ob es sich wirklich um ein Design-Problem handelt, oder es eine gute Implementierung ist.

3.4.2.4 Phase 4: Detaillierte Beschreibung des Systems (Detailed Model Capure)

In diesem Schritt wird eine den Zielen des Projekts entsprechende Beschreibung des Systems erstellt. Zusätzlich sollen die versteckten Artefakte im Code gefunden werden, die vorher noch nicht erkannt wurden. Die Informationen aus Phase 3 legen das Fundament dafür. In Phase 4 gibt es vier grundsätzliche Problempunkte:

- *Details matter:* Details sind im Software-Engineering oft wichtig. Es fällt aber schwer, die wichtigen Details auszumustern.
- *Details remains implicit:* Design-Entscheidungen sind oft nicht explizit erklärt. Dadurch fällt es schwer nachzuvollziehen, warum etwas genau so gelöst wurde.
- *Design does evolve:* Software verändert sich während ihrer Lebenszeit ständig (siehe Kapitel Software-Evolution). Dadurch sind die Design-Dokumente

natürlich nie aktuell und reflektieren den aktuellen Stand des Systems. Deswegen ist es sehr informativ zu wissen, warum und inwiefern sich die Software verändert hat.

- *Static structure versus Dynamic behaviour:* Während einer statischen Analyse kann die Struktur und die Hierarchie des Programms festgestellt werden. Viel schwieriger wird es aber herauszufinden, welche Interaktionen während der Laufzeit vorhanden sind.

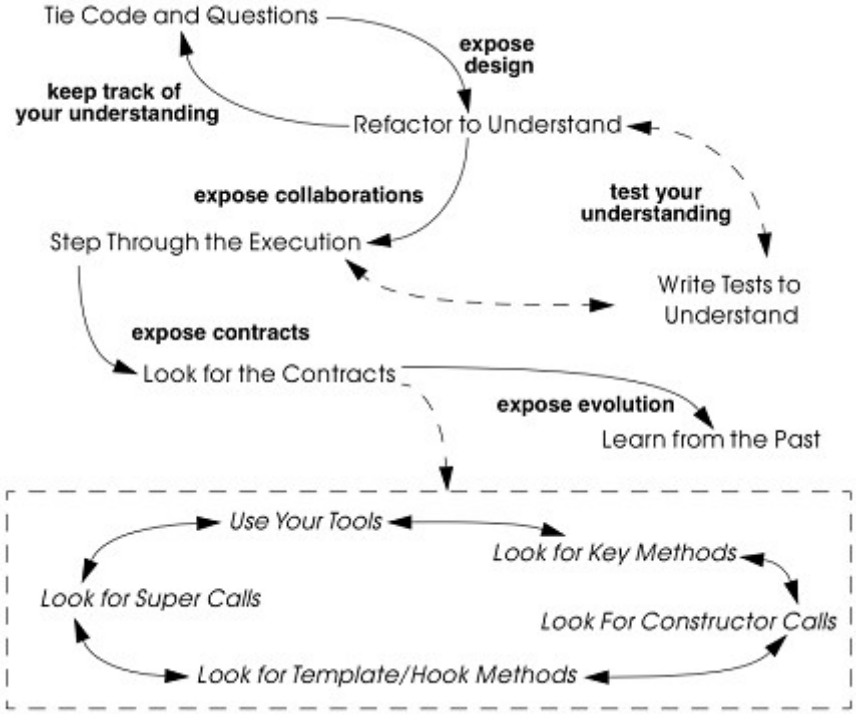

Abbildung 10: Aufgaben in Schritt 4

Quelle: [DeDuNi08] unter der Lizenz http://creativecommons.org/licenses/by-sa/3.0/

Phase 4 ist etwas umfangreicher. Jegliche Diskrepanzen und offene Punkte beim Durchsehen des Codes sollen direkt kommentiert werden (*Tie Code and Questions*). Dadurch werden diese Informationen nicht vom Kontext getrennt. Durch Veränderung des Quellcodes dahingehend, dass eine neue, testweise Funktionalität hinzugefügt wird, kann einiges über das Legacy-System erfahren werden (*Refactor to Understand*). Angenommen, es soll die Kapazität eines Datenstruktur von 1024 Bytes auf 2048 Bytes erhöht werden. Durch vorherige Reverse Engineering-Maßnahmen wurde herausgefunden, dass im inneren die Datenstruktur im Inneren nur aus einem Byte-

Array besteht. Die Länge des Arrays wird nun auf 2048 erhöht. Nach einem Testlauf *(Write Tests to Understand*) kommt es zu einem Datenbankfehler. Offensichtlich gibt es eine Verbindung zwischen dieser Datenstruktur und der Datenbank. Beim Benützen des Debuggers (*Step Throug the Execution*) wird erkannt, wo sich die Data Access Objects befinden, welche die Daten persistieren.

Die Hauptaufgabe von Interfaces ist es, Zugriffe von Außen zu ermöglichen. Wenn die Funktion der Interfaces nicht ganz klar ist, können über ihre Verwendung Rückschlüsse auf ihre Aufgabe gemacht werden (*Look for the Contracts*). Ein Contract ist ein Begriff aus dem Prinzip „Design by Contracts“ [Me97]. Interfaces definieren dabei einen Vertrag (Contract), an den sie sich halten. Abweichungen von diesem Vertrag sollen mittels eines Fehlers quittiert werden. Dafür werden bei korrekten Eingabewerten wohldefinierte Verhaltensweisen garantiert. In Abbildung 10 befasst sich das untere Kästchen mit möglichen Schritten, um solche Contracts herauszufinden.

Neben den offensichtlichen Vorteilen von Versions-Control-Systemen ermöglichen sie bei einem Reverse Engineering-Projekt den Vergleich historisch unterschiedlicher Versionen *(Learn from the Pas*t). Die Differenzen zwischen Versionen und die Log-Statements von Checkins sind ein Quell an Informationen über das System und seine Umgebung.

4 Analyse von AR-Applikationen

Um herauszufinden, wie die aktuellen Barrieren lauten, wurde eine Evaluation der derzeitigen semi-automatisierten AR-Tool-Landschaft gemacht.

4.1 QSOS (Qualification and Selection of Open Source software)

Im Dokument „*Method for Qualification and Selection of Open Source software*“[Atos06] wird das QSOS-Framework beschrieben. Es dient zur Evaluation von Open-Source-Software und wird von der Firma Atos Origin[3] entwickelt. Veröffentlich ist es unter der GNU Free Documentation License 1.2[4]. Die Version 1.3 des Frameworks wurde im Jahr 2004 released und war die erste, öffentlich zugängliche Fassung. Zum Zeitpunkt dieser Arbeit, ist die Version 1.6 aus dem Jahr 2006 aktuell. Auf dieser basiert die durchgeführte Evaluation.

In QSOS gibt es vier Schritte die, einen iterativen Prozess bilden.

3 http://www.atosorigin.com/

4 http://www.gnu.org/licenses/fdl-1.2.html

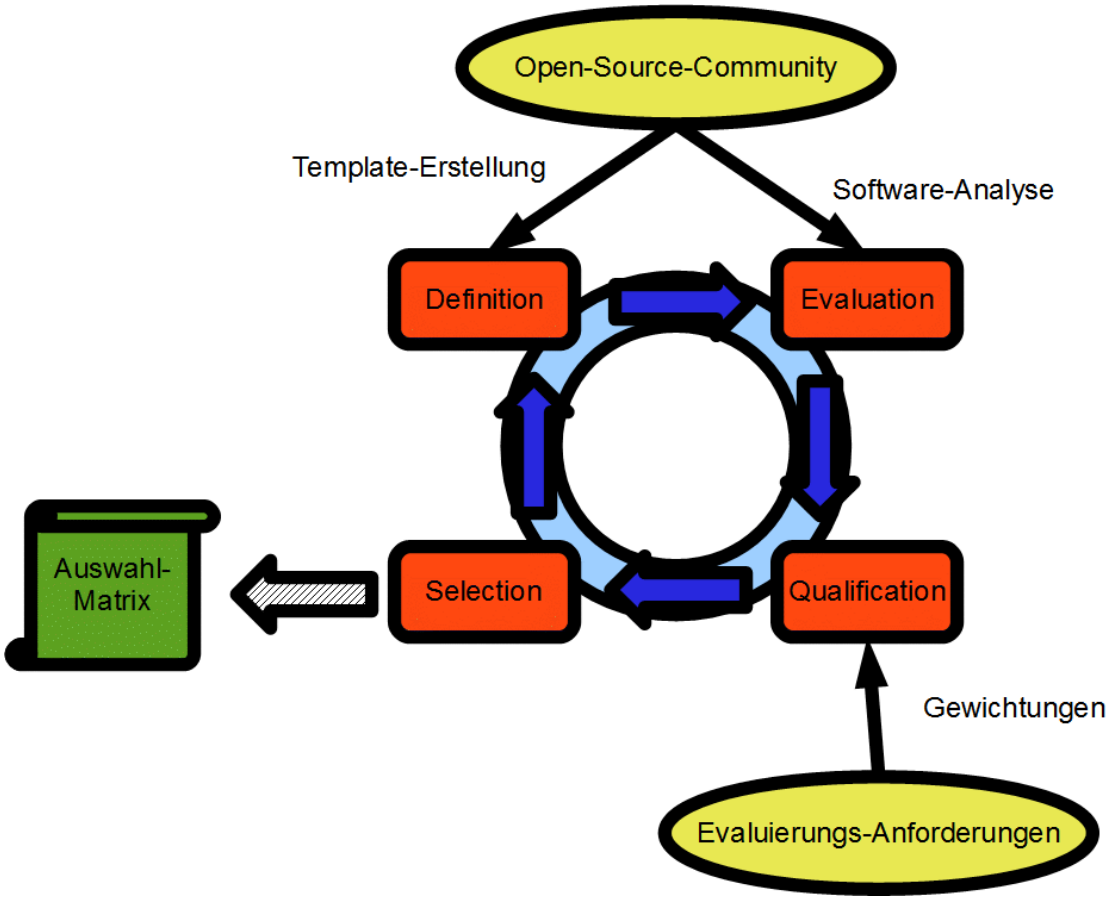

Abbildung 11: Der QSOS-Prozess

4.1.1 Definition

In dieser Phase werden sogenannte „Frames“ generiert. Dies sind die Bewertungskriterien der zu untersuchenden Software. Es gibt bereits einige vorgefertigte Frames, es können aber natürlich auch eigene definiert werden (Tailoring).

Für diesen Schritt steht ein ein Template-Editor zur Verfügung, der dem Framework beiliegt.

4.1.2 Evaluation

Für jede zu untersuchende Software wird eine Identity-Card ausgefüllt, die grundsätzliche Informationen über die Software enthält.

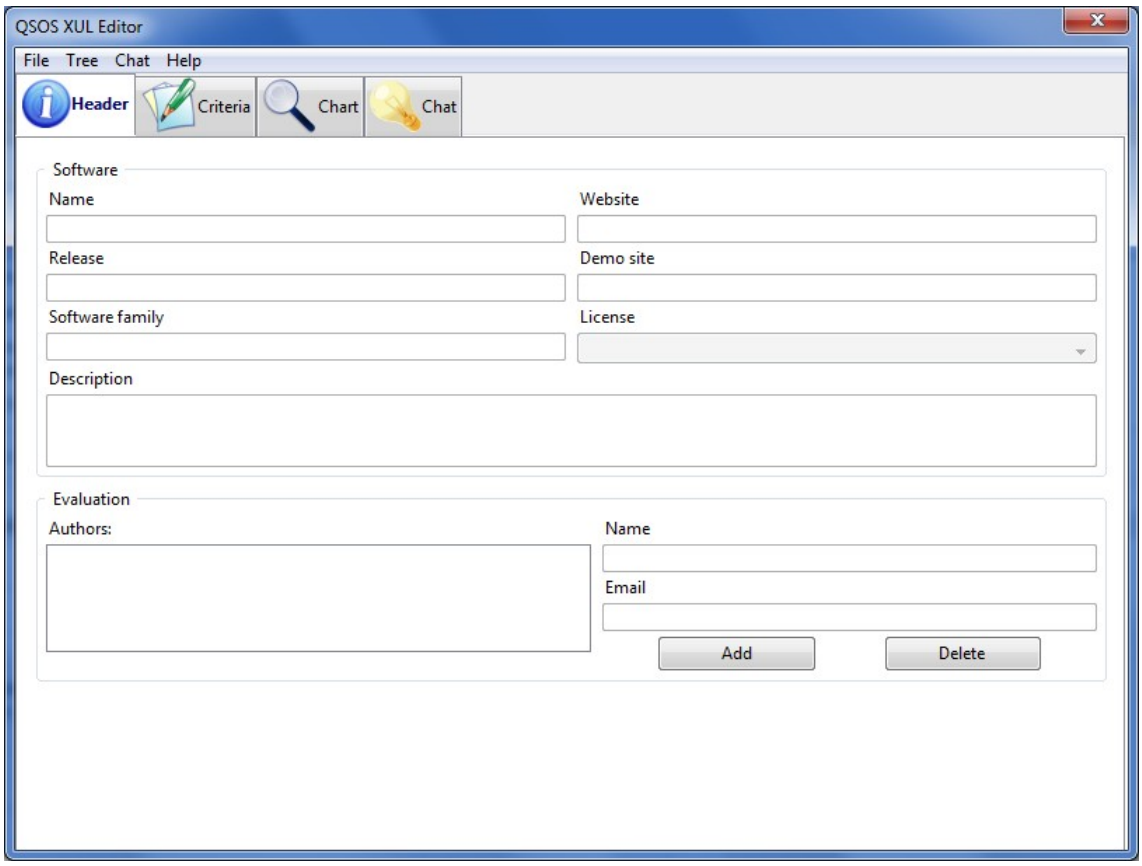

Abbildung 12: Eine Identity-Card im Editor

Weiters werden natürlich die definierten Frames ausgefüllt. Zur Bewertung gibt es drei mögliche Werte.

Bewertung	Funktionalitäts-Anforderung erfüllt
0	Nicht erfüllt
1	Teilweise erfüllt
2	Komplett erfüllt

Tabelle 1: Bewertungsschema in QSOS für ein Criterion

Dabei werden immer nur die unterste Stufe der Bewertungskriterien evaluiert. Die Einstufung für die höheren Levels erfolgt mittels eines gewichteten Mittelwerts der niedrigen Stufen. Ein Beispiel einer solchen Kriterien-Aufteilung kann in Abbildung 13 gesehen werden.

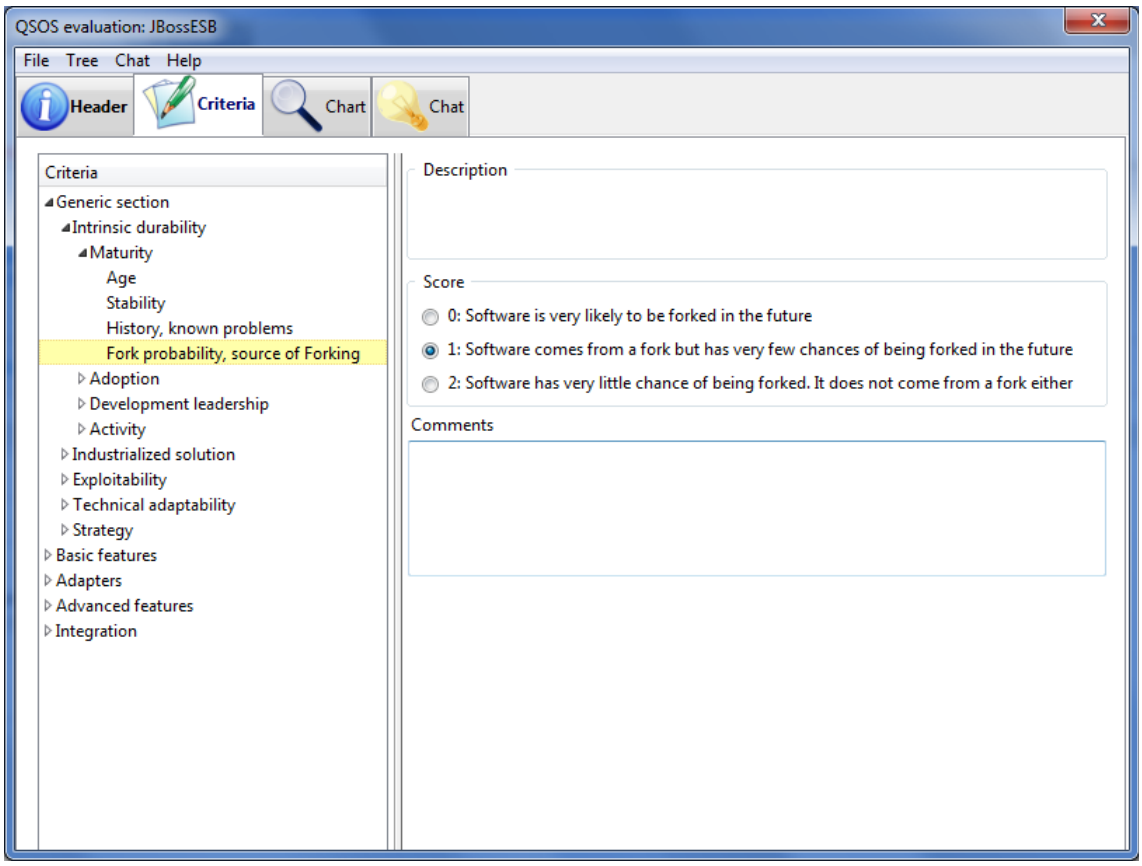

Abbildung 13: Aufteilen der Kritierien in kleine Teile am Beispiel JBossESB

Der Durchschnitt aus dem Ergebnis von *Age, Stability, History* und *Fork propbability* liefert die Auswertung für *Maturity*.

4.1.3 Qualification

In dieser Phase erfolgt die Gewichtung der einzelnen Kriterien. Damit ist es möglich eine sinnvolle Bewertung für den benötigten Kontext herzustellen. In QSOS werden diese Gewichtungen Filter genannt.Vier verschiedene Filter stehen zur Verfügung:

- *Filter auf die Identity-Card:* Es kann zum Beispiel bestimmt werden, dass nur Software mit einer gewissen Lizenz weiter betrachtet wird.
- *Filter auf die Funktionalität:* Es kann angegeben werden, ob eine Funktionalität optional, benötigt oder nicht benötigt wird.
- *Filter auf die User-Risiken:* Gleich wie bei den Funktionalitäten können hier die Risiken gewichtet werden.

- *Filter auf Service-Provider-Risiken:* Dieser Filter kann von Service-Provider verwendet werden, um die Integrierbarkeit und den Lock-In zu gewichten.

4.1.4 Selection

In diesem Schritt wird die Software identifiziert, welche die gewünschten Kriterien erfüllt. Natürlich ist es auch hier möglich, Software aus der gleichen Familie miteinander zu vergleichen. Es gibt zwei Arten, eine *Selection* durchzuführen.

4.1.4.1 Strict selection

Bei dieser Art werden die Filter aus dem *Qualification*-Schritt rigide angewendet. Software, die eines der benötigten Kriterien nicht erfüllt, wird nicht in die Selektion aufgenommen. Es kann dabei natürlich vorkommen, dass keine untersuchte Software die Anforderungen erfüllt. Software die es in die Selektion geschafft hat, bekommt eine Punkteanzahl verliehen, welche aus der Gewichtung berechnet wird.

4.1.4.2 Loose selection

Hier wird Software die laut Filter nicht erwünscht ist, nur als solche klassifiziert. Die dadurch entstandenen Bewertungs-Lücken, werden durch Werte aus den Filtern aufgefüllt.

4.1.4.3 Gewichtung

Folgende Kriterien dienen zur Bewertung von Anforderungen und Risiken:

Wichtigkeit der Funktionalität	Gewichtung
Muss-Funktionalität	+3
Optionale Funktionalität	+2
Nicht benötigte Funktionalität	0

Tabelle 2: Criterion-Gewichtung nach Funktionalität

Relevanz des Risikos	Gewichtung
Kritisches Risiko	+3 oder -3
Wichtiges Risiko	+1 oder -1
Irrelevantes Risiko	0

Tabelle 3: Criterion-Gewichtung nach Risiko

Wie man erkennen kann, gibt es bei der Gewichtung auch negative Werte. Es können also auch negative Gewichtungen angewendet werden. Dadurch ist es möglich, nicht gewünschte Criterions auszudrücken.

4.1.4.4 Vergleich

Wie bereits erwähnt ist der Vergleich ähnlicher Software miteinander sehr einfach. Das Tool O3S[5] führt Selektion und Bewertung automatisch durch. Zusätzlich können Vergleiche komfortabel als Netzdiagramm abgebildet werden.

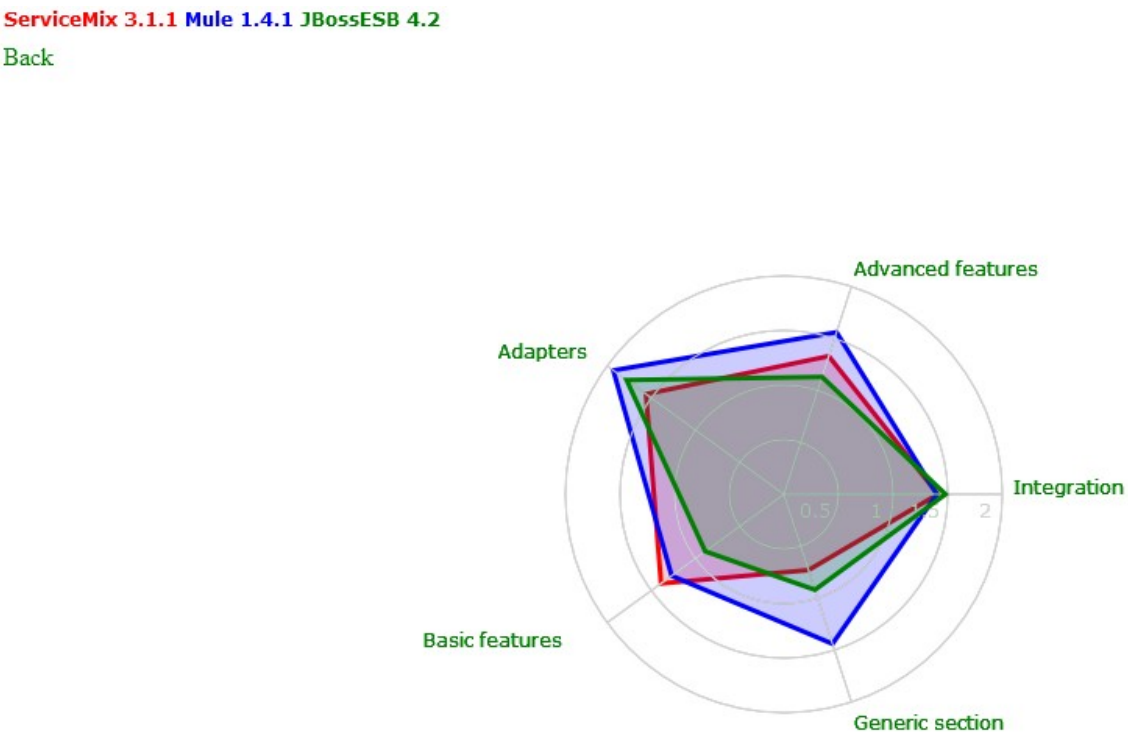

Abbildung 14: O3S-Plot eines Vergleichs zwischen Enterprise Service Bus

Abbildung 14 zeigt einen derartigen Vergleich zwischen drei ESB-Applikationen.

5 http://www.qsos.org/o3s/

4.2 Anforderungen an die Target-Software

Folgende Anforderungen wurden an die Target-Software gestellt.

- *Komplexität: Klassenanzahl 100 bis 500*
  Die Anwendung soll komplex aber doch übersichtlich bleiben.
- *Java*
  Java hat sich als Programmier-Sprache im akademischen Bereich durchgesetzt und die meisten Methoden-Implementierungen unterstützen es.
- *Software soll bereits bekannt sein*
  Wenn die Target-Software nicht bereits bekannt ist, würde sich von einer evaluierten Anwendung zur nächsten die Ausgangsbedingungen zu stark ändern.
- *Gute Dokumentation soll vorhanden sein um einen sinnvollen Test zu machen*
  Dadurch ist bereits eine Referenz vorhanden, gegen welche die untersuchten Tools antreten müssen.
- *SVN soll als Quellcode-Verwaltung verwendet werden*
  Dies ist eine oft eingesetzte Quellcode-Verwaltung.
- *CVS soll als Quellcode-Verwaltung verwendet werden*
  Dies ist eine veraltete Quellcode-Verwaltung welche bei Legacy-Projekten noch im Einsatz sein kann. Da ein AR-Projekt oft für Legacy-Anwendungen gemacht wird, soll die CVS-Fähigkeit auch untersucht werden.

Diese Anforderungen konnten nicht von einer Software erfüllt werden. Der Großteil der Criterions wurde mittels XStream überprüft. Für spezielle Java-Features und den CVS-Support wurde jeweils eigene Projekte verwendet.

4.2.1 Target-Software XStream

XStream ist eine Java-Bibliothek, die Java-Objekte in XML serialisieren kann. Zusätzlich hat man auch die Möglichkeit, in XML gespeicherte Objekte wieder zu deserialisieren. Objekte, die Member eines zu speichernden Objekts sind, werden ebenfalls serialisiert. Der Objektbaum unterhalb des Root-Objekts wird also komplett aufgelöst. Die Software unterstützt auch andere Speicherformate wie JSON [Crock06].

Als Basis für die Evaluation wird der Trunk[6] mit der Revision 1847 verwendet.

4.2.1.1 Verwendungsbeispiele

Ein Objekt in seine XML-Repräsentation umzuwandeln funktioniert sehr einfach:

Beispielklasse:

```
public class Vehicle {
      private Engine installedEngine;
      private String manifacturer;
      private String modelName;
      private double price;
      //..constructors & accessors
}
public class Engine{
      private int horsePower;
      private double consumption;
      private String engineName;
      //..constructors & accessors
}
```

```
Engine myEngine = new Engine(90,5.6,"Mark I");
Vehicle myVehicle = new Vehicle(myEngine, "TU Wien",
                                "Spacer 3", 30000);
XStream xStreamWorker = new Xstream();
String xmlRepresentation = xStreamWorker.toXML(myVehicle);
```

Das Objekt ist nun in der Variable *xmlRepresentation* als XML-Text gespeichert. Mittels des folgenden Aufrufs kann aus dem XML-Text wieder ein Objekt erzeugt werden:

6 XStream-SVN-Repository: http://svn.codehaus.org/xstream/trunk

```
Vehicle loadedVehicle = (Vehicle) xStreamWorker.fromXML(
                                                xmlRepresentation);
```

Die Variable *loadedVehicle* beinhaltet jetzt das deserialisierte Objekt.

4.2.1.2 Architektur von XStream

XStream ist in vier Komponenten aufgeteilt:

- Converters
- Drivers
- Context
- Facade

4.2.1.2.1 Converter

Converter wandeln ein Objekt in XML um oder machen diesen Vorgang rückgängig. Es gibt für oft verwendete Typen bereits mitgelieferte Converter. Sollte es für eine Klasse keinen Converter geben, wird standardmäßig der Default-Converter verwendet. Es ist natürlich möglich, weitere Converter hinzuzufügen (Extensibility).

4.2.1.2.2 Drivers

Drivers verbinden XStream mit XML-Readern und -Writern. Durch diese Abstraktionsschicht ist es möglich, beliebige XML-Frameworks zu verwenden.

4.2.1.2.3 Context

Der Context beschreibt den Objektgraphen, der (de)serialisiert werden soll. Der Context ruft die nötigen Converter auf, um den Objektgraphen zu serialisieren.

4.2.1.2.4 Facade

Als Facade wird die Klasse XStream bezeichnet. Sie bietet einen einfacheren Zugriff auf die Funktionalität von XStream. Die Klasse XStream wendet also das Entwurfsmuster Fassade an.

4.2.2 Test-Projekt für Generics

Für die Überprüfung, ob die untersuchte Software Generics unterstützt, wurde ein einfaches Java-Projekt mit zwei Klassen erstellt. Es soll dadurch untersucht werden, ob die Applikation die Abhängigkeit erkennt. Der Quellcode der zwei Klassen sieht folgendermaßen aus:

```
package genTextUsed;

public class MyGenericType {

	public void Print() {

		System.out.println("Test");

	}

}
```

```
package genTextUser;

import genTextUsed.MyGenericType;

public class MyGenericsUser  <T extends MyGenericType>{

	public String GetTest() {

		return "Test";

	}


	public T GetGeneric(T gen) {

		gen.Print();

		return gen;

	}

}
```

Daraus lässt sich das Dependency-Diagramm aus Abbildung 15 ableiten.

Abbildung 15: Abhängigkeits-Diagramm des Generics-Test-Projekts

Im Optimalfall soll diese Abhängigkeit erkannt und auch dementsprechend verarbeitet werden.

4.2.3 Test-Projekt für Annotations

Um zu Überprüfen, ob das Java-Feature Annotations unterstützt wird, wurde ebenfalls ein Test-Projekt erstellt. Das Projekt besteht aus zwei Klassen.

```
package annoUsed;

public @interface CustomAnnotation {

	String Name();

}
```

```
package annoUser;

import annoUsed.CustomAnnotation;

@CustomAnnotation(Name = "Test")

public class AnnotationUser {

	public String GiveText() {

		return "Text";

	}

}
```

Hier muss die Abhängigkeit, wie sie in Abbildung 16 gezeigt wird, erfasst werden.

Abbildung 16: Abhängigkeit im Tes-Projekt für Annotations

Sollte dies nicht der Fall sein, wird das Criterion Annotations mit null Punkten bewertet.

4.2.4 Test-Projekt für Dependency-Injection (Spring)

Für den Test auf die Unterstützung von Dependency-Injection wurde das Framework Spring verwendet, da es im Java-Umfeld das am meisten verwendete Library für die Realisierung der Dependency-Injection ist. Da es sich hierbei um kein implizites Feature der Sprache Java handelt, gestaltet sich der Aufbau des Projekts etwas komplexer. Es gibt zwei Klassen, ein Interface und eine XML-Datei, wie aus Abbildung 17 ersichtlich ist.

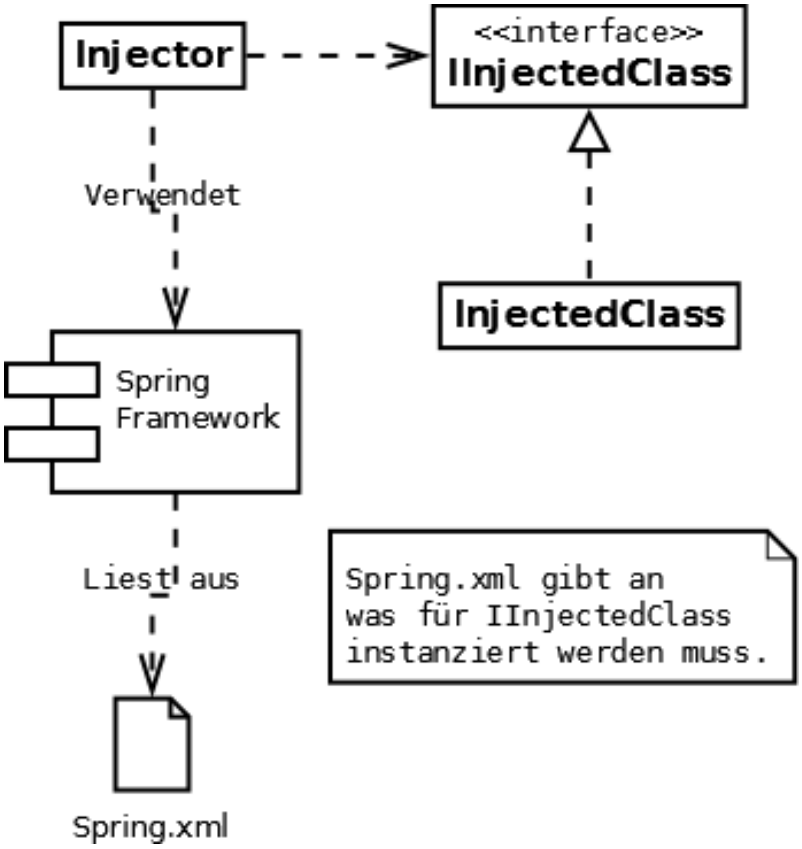

Abbildung 17: Aufbau des Spring-Test-Projekts

Weiters wurden dem Projekt natürlich auch Referenzen zum Spring-Framework

hinzugefügt. Der Quellcode der drei Klassen ist unterhalb aufgelistet.

```
package springUsed.interfaces;

public interface IInjectedClass {

	public void HelloWorld();

}
```

Die Klasse *springUsed.InjectedClass* implementiert, wie man sehen kann, das Interface *springUsed.interfaces.IInjectedClass*.

```
package springUsed;

import springUsed.interfaces.IInjectedClass;

public class InjectedClass implements IInjectedClass {

	/* (non-Javadoc)
	 * @see springUsed.IInjectedClass#HelloWorld()
	 */
	@Override
	public void HelloWorld() {
		System.out.println("Hello World");
	}

}
```

Die Klasse *springUser.Injector* verwendet das Interface *springUsed.interfaces.IInjectedClass*. Mittels Spring wird eine Instanz einer Implementierung von *springUsed.interfaces.IInjectedClass* erzeugt. Zu beachten ist, dass aus dem Source-Code nicht ersichtlich ist, dass *springUser.Injector* eigentlich eine Abhängigkeit mit *springUsed.InjectedClass* hat. Diese ist nämlich in der Konfigurationsdatei von Spring definiert.

```
package springUser;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
```

```
import springUsed.interfaces.IInjectedClass;

public class Injector {

      public static void main(String[] args) {

            XmlBeanFactory beanFactory = new XmlBeanFactory(

                        new ClassPathResource("spring.xml"));

            IInjectedClass myBean = (IInjectedClass) beanFactory

                                    .getBean("IInjectedClass");

            myBean.HelloWorld();

      }

}
```

Im Optimal-Fall sollen die AR-Tools erkennen, dass eine Abhängigkeit vorliegt, oder darauf hinweisen, dass es eine nicht offensichtliche Abhängigkeit gibt.

4.2.5 org.eclipse.compare – Überprüfung der Software-Evolution bei CVS

Dieses Projekt dient der Feststellung, ob die zu evaluierende Anwendung Software-Evolution mit CVS unterstützt. Da XStream als Quellcode-Verwaltung SVN verwendet, muss ein anderes Projekt herangezogen werden. In diesem Fall ist es das Eclipse-Plugin *org.eclipse.compare*, welches standardmäßig mit Eclipse ausgeliefert wird.

4.3 Erstellung eines Evaluations-Templates

Für die Evaluation der ausgewählten Reverse Engineering-Tools wurde das QSOS-Framework ausgewählt. Als erster Schritt wird ein Template erstellt, welches die Basis des Assessment der zu untersuchenden Software darstellt. In der QSOS-Terminologie ist dies der Schritt *Definition*.

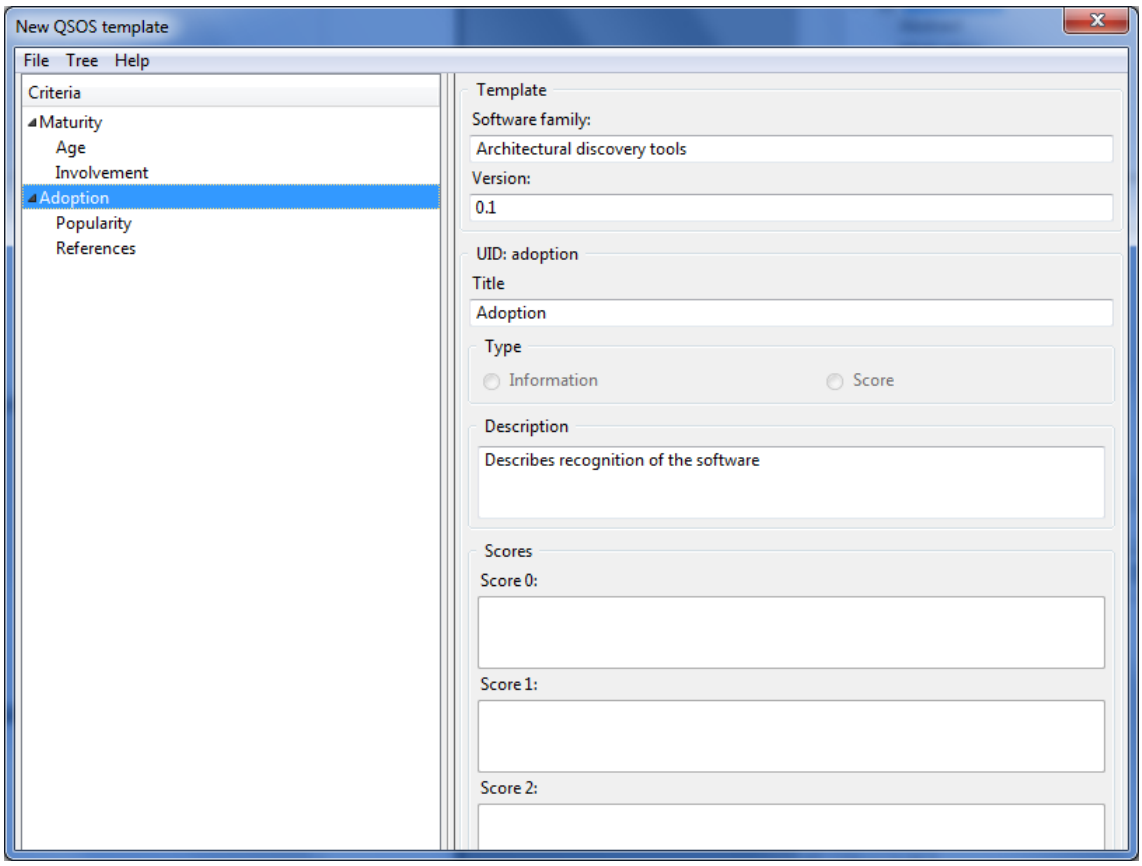

Abbildung 18: QSOS-Template-Editor

Als Basis wurden bereits fertiggestellte QSOS-Templates verwendet und diese einem Tailoring-Prozess unterzogen. Der von Raphael Semeteys entwickelte „QSOS Template XUL Editor“ wurde als Unterstützung beim Erstellen der Vorlage verwendet. Da angestrebt wird, dass die Ergebnisse der Auswertung in die QSOS-Datenbank aufgenommen werden, sind die Criterions in Englisch erfasst.

4.3.1 Criterions

In diesem Kapitel sind die verwendeten Criterions aufgelistet, die für die Evaluation der Tools durch das QSOS-Framework bestimmt wurden. Zu beachten ist, dass Titel, Beschreibung und Bewertungsrichtlinien auf Englisch verfasst sind. Dadurch ist es möglich, dass das Bewertungs-Template für AR-Tools einem größeren Benutzerkreis zugänglich wird. Eine detaillierte Auflistung der Criterions ist im Appendix unter QSOS-Template auf Seite 84 zu finden. Folgend werden die Beweggründe für die Wahl der Kategorien und ihres Inhalts erklärt.

Wie in der QSOS-Beschreibung vorgeschlagen, wurden mehrere Iterationen ausgeführt,

um eine Gefühl dafür zu bekommen, ob die Evaluation mit diesen Parameter überhaupt den gewünschten Effekt hat.

Aus dem erstellten QSOS-Template wurde mittels Perl eine leeres Evaluations-Sheet erstellt.

4.3.1.1 Generic section

Diese Kategorie befasst sich mit der Umgebung der Applikation. Es werden Faktoren wie Produkt- und Projektmanagement bewertet. Eine Bewertung der Umgebung macht Sinn, da diese direkten und indirekten Einfluss auf die Applikation hat. Angenommen, das letzte Release ist fünf Jahre alt und die Aktivität im User-Forum ist niedrig, kann davon ausgegangen werden, dass Support kaum vorhanden ist.

Weiters ist es wichtig zu wissen, welche Verbreitung das Tool bereits hat. Je größer die Verbreitung, desto eher kann mit Hilfe gerechnet werden. Ein Open-Source-Programm, das eine starke Verbreitung hat, wird auch eher Bug-Fixes und neue Features erhalten.

Eine nicht unwesentliche Information sind auch die unterstützten Systemumgebungen. Ein Reverse Engineering eines Unix-Tools wird mit einem AR-Tool, das nur unter Windows läuft, nur schwer möglich sein.

Relevant für die Verwendung der Applikation sind auch die Lizenz und die Verfügbarkeit des Source-Codes.

4.3.1.2 User interface

User interface befasst sich mit der Internationalisierung der Applikation. Gerade außerhalb des englischsprachigen Raumes ist dies sinnvoll zu überprüfen. Die Möglichkeit, das Aussehen des Tools seinen Wünschen anzupassen wird ebenfalls bewertet.

4.3.1.3 Prerequisite

Diese Kategorie hat reinen Informationscharakter. Sie hält fest, in welcher Sprache die Applikation entwickelt wurde. Dies ist dann wesentlich, wenn Erweiterungen oder Bug-Fixes durchgeführt werden müssen. Die Moose Tool Suite verwendet beispielsweise Smalltalk als Entwicklungssprache. Dies kann unter Umständen zu Ressourcen-

Engpässen führen.

4.3.1.4 Storage support

In *Storage support* wird festgehalten, welche Möglichkeiten der Datenspeicherung vorhanden sind. Möglichkeiten der Datenspeicherung sind dabei:

- Dateien
- Datenbanken
- Versionsverwaltungssysteme
- Data Warehouse

4.3.1.5 GUI

Diese Kategorie beschreibt die Möglichkeit, die Applikation in bestehende Entwicklungsumgebungen einzubinden. Wegen einer Einschränkung der QSOS-Tools konnte diese Sektion nicht mit User interface gemerged werden. Deshalb wurde ein eigener Punkt angelegt.

4.3.1.6 Data

Dieser Punkt befasst sich mit den Möglichkeiten, Daten zu ex-/importieren. Im Optimalfall können die Daten aus dem Source-Code und den Kompilaten extrahiert werden. Weiters soll der Export in ein offenes und bekanntes Format wie etwa FAMIX [Gir11] erfolgen.

Es soll auch aufgelistet werden, welche Programmiersprachen das Tool als Input-Daten unterstützt. Zu beachten ist, dass dies nicht das Gleiche wie Punkt Prerequisite ist.

4.3.1.7 Architectural recovery methods

Hier wird bewertet, welche Methoden des Architectural Recovery unterstützt werden. Dabei fanden ausgewählte Methoden aus dem Kapitel AR-Methoden Verwendung.

4.3.1.8 Visualization

Dieser Punkt bewertet das Vorhandensein und die Unterstützung verschiedener

Software-Visualisierungen. Die folgenden Abbildungs-Verfahren, aus dem Kapitel Visualisierungsarten, wurden überprüft:

- Dendency-Diagramme
- Tree-Maps
- Polymetric Views
- UML-Diagramme
- Call-Graph-Visualisierungen

Des Weiteren gilt es festzustellen, ob es möglich ist die Illustrationen sofort zu dokumentieren. Im Optimalfall können Kommentare zum Diagramm hinzugefügt werden.

4.3.1.9 Java feature support

Hier soll überprüft werden, ob oft eingesetzte Java-Features, welche nach Java 1.4 eingeführt worden sind, unterstützt werden. Dies sind im Konkreten *Generics* und *Annotations*. Weiters ist es noch relevant, ob *JavaDocs* bei der Analyse oder Präsentation berücksichtigt werden.

Es kann vorkommen, dass Legacy-Systeme einen *Dependency-Injection-Container* wie Spring einsetzen. Deshalb muss auch überprüft werden, ob dies unterstützt wird. Die Untersuchung soll mit Spring vollführt werden.

4.4 Ergebnisse der Evaluation

Bei der Evaluation wurden jeweils Gewichtungen (siehe Kapitel Gewichtung) der Stärke eins (1) verwendet. Zu beachten ist, dass zwei (2) jeweils der Maximum-Wert ist. Null (0) ist das Minimum.

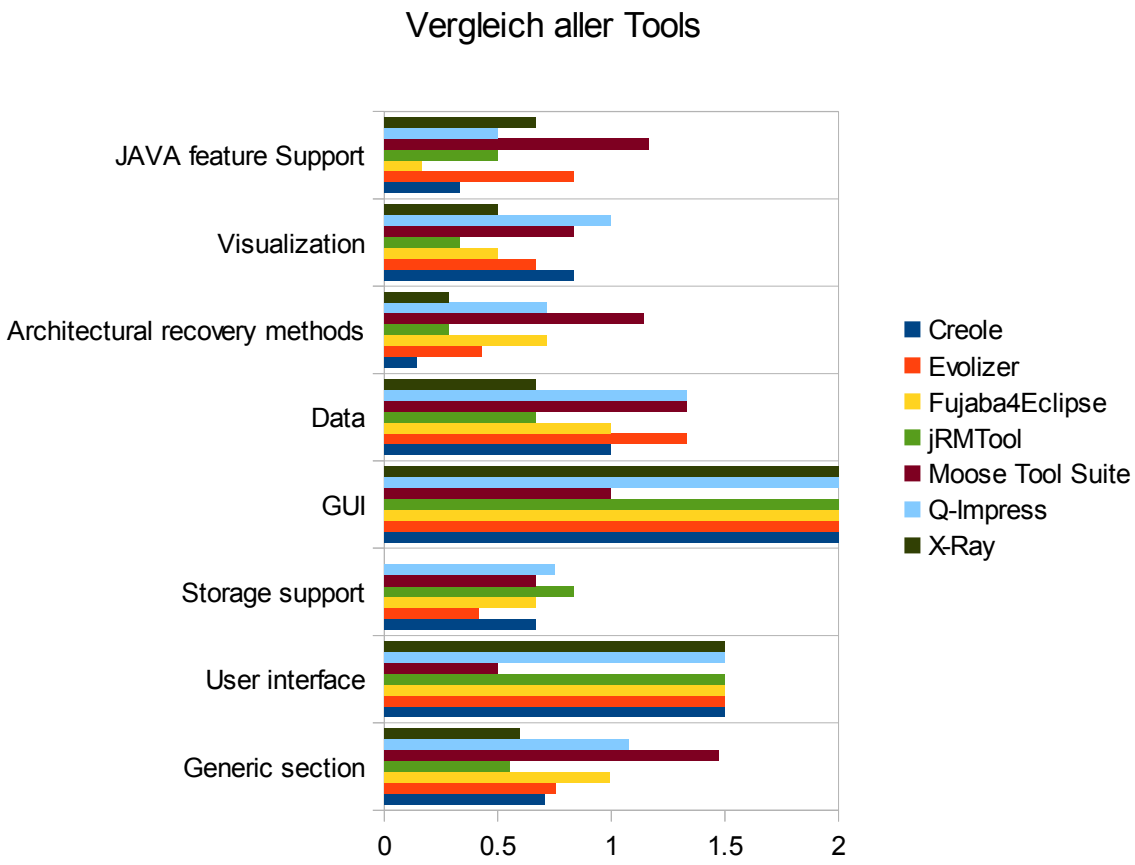

Abbildung 19: Vergleich aller evaluierten Tools

In Abbildung 19 ist der Vergleich aller Tools und ihre Ergebnisse innerhalb der einzelnen Sektion zu finden.

4.4.1 Evolizer

Evolizer ist eine Plattform zur Analyse von Software-Evolution. Als Versions-Control-System wird derzeit CVS unterstützt. Der Kern besteht nur aus den Daten-bereitstellenden Komponenten. Es werden zwei Addons für Evolizer in Verbindung untersucht, welche die gesammelten Daten visualisieren. Dies sind *DA4Java* und *SNAMap*.

DA4Java ist die Abkürzung für *Dependency Analysis for Java*. Mit diesem Addon können die Abhängigkeiten zwischen den Programm-Komponenten untersucht werden. Dies wird durch den Einsatz von *Polymetric Views* erreicht.

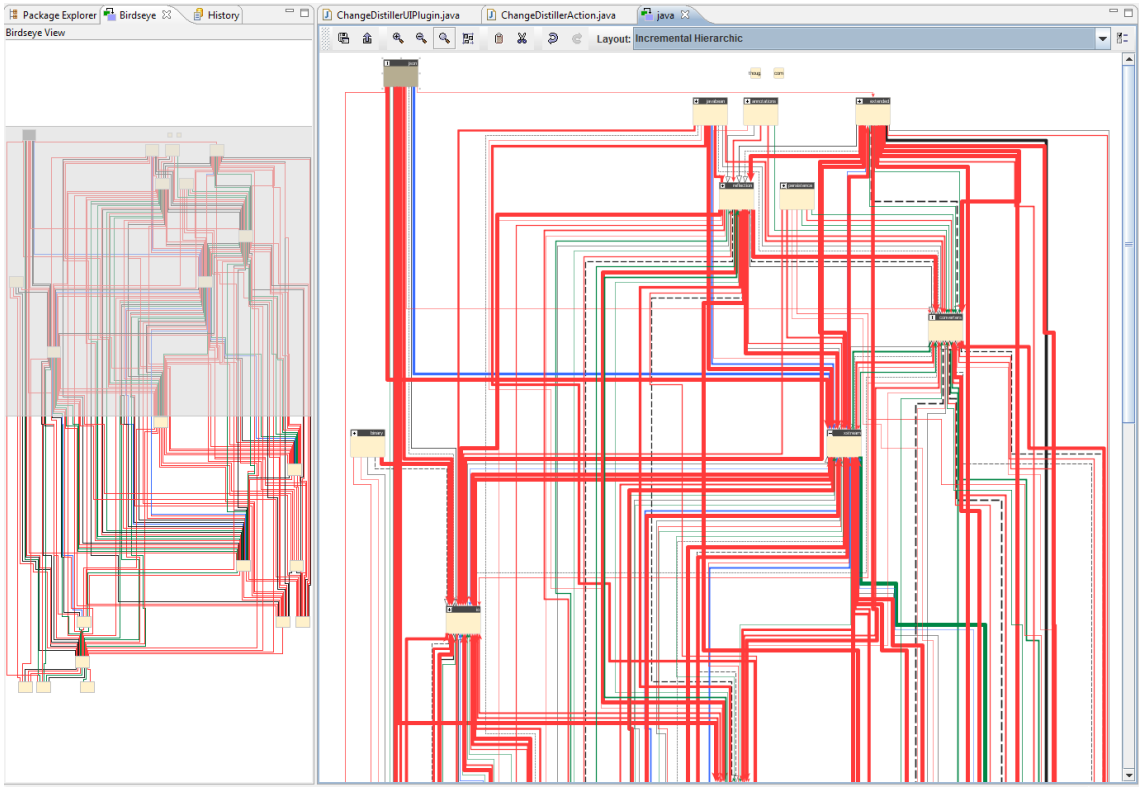

Abbildung 20: XStream (ohne Unit-Tests) dargestellt in DA4Java

Der *SNAnalyzer (Developer Contribution Analyzer*) kümmert sich um die Visualisierung von Änderungen innerhalb eines Zeitraums.

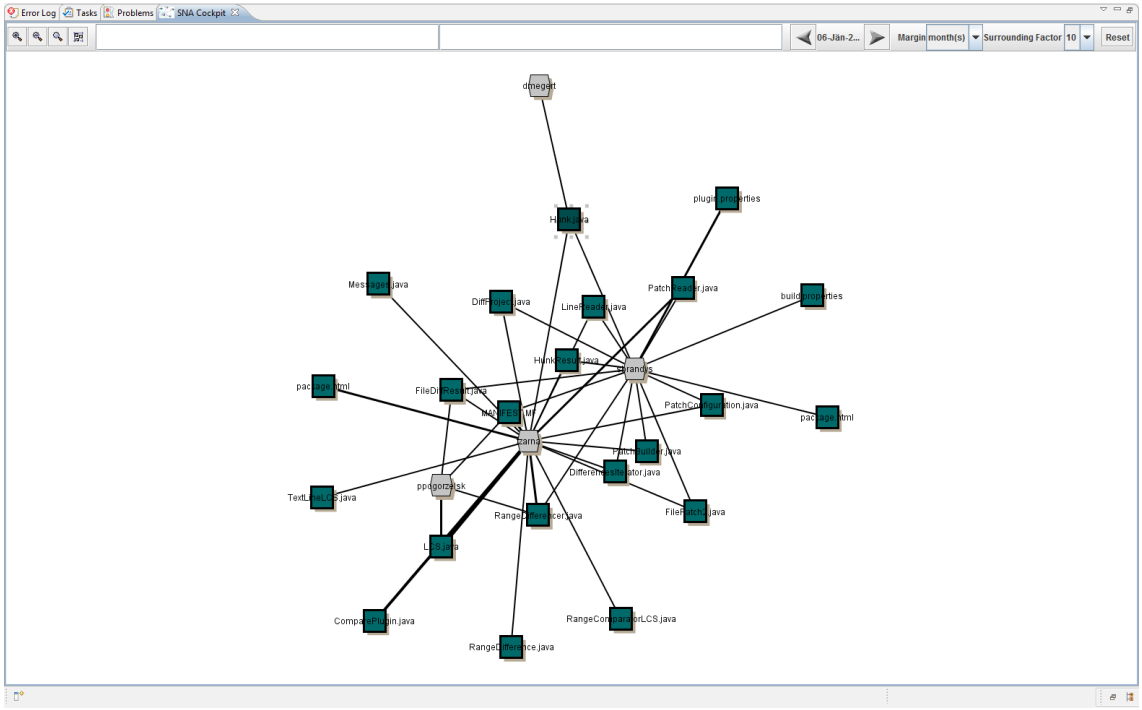

Abbildung 21: org.eclipse.compare.core dargestellt in SNAnalyzer

In Abbildung 20 sind grüne und graue Nodes zu sehen. Die geänderten Komponenten sind grün dargestellt. Bei den grauen Kästchen handelt es sich um Entwickler. Durch die Verbindungen ist ersichtlich, welche Entwickler im untersuchten Zeitraum welche Komponenten geändert haben.

4.4.1.1 Evaluationsergebnis Evolizer

Abbildung 22 zeigt das Evaluationsergebnis. Evolizer verwendet das Datenformat der Moose Tool Suite – FAMIX. Dies bringt einen erheblichen Vorteil bei der Interoperabilität der beiden Programme.

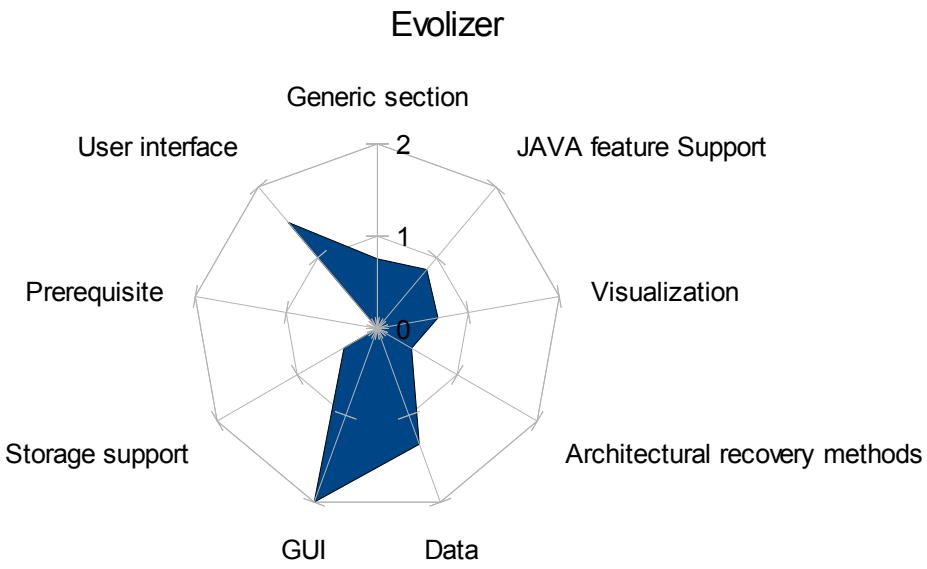

Abbildung 22: Analyse-Ergebnisse Evolizer

Wie aus dem Namen hervorgeht spezialisiert sich Evolizer auf die AR-Methode *Software-Evolution*. Für dieses Feature bietet Evolizer nur das Backend an. Die Plugins DA4Java und SNAnalyzer visualisieren diese gewonnenen Daten. Beim Dependency-Diagramm ist es möglich, die dahinterliegenden Metriken selbst zu definieren. Dependency-Diagramm und Polymetric View sind also kombiniert vorhanden.

4.4.2 Creole/SHriMP

Mit SHriMP lässt sich Source-Code visuell erforschen. Creole ist dabei das Eclipse-Plugin, welches sich auf Java-Code spezialisiert hat.

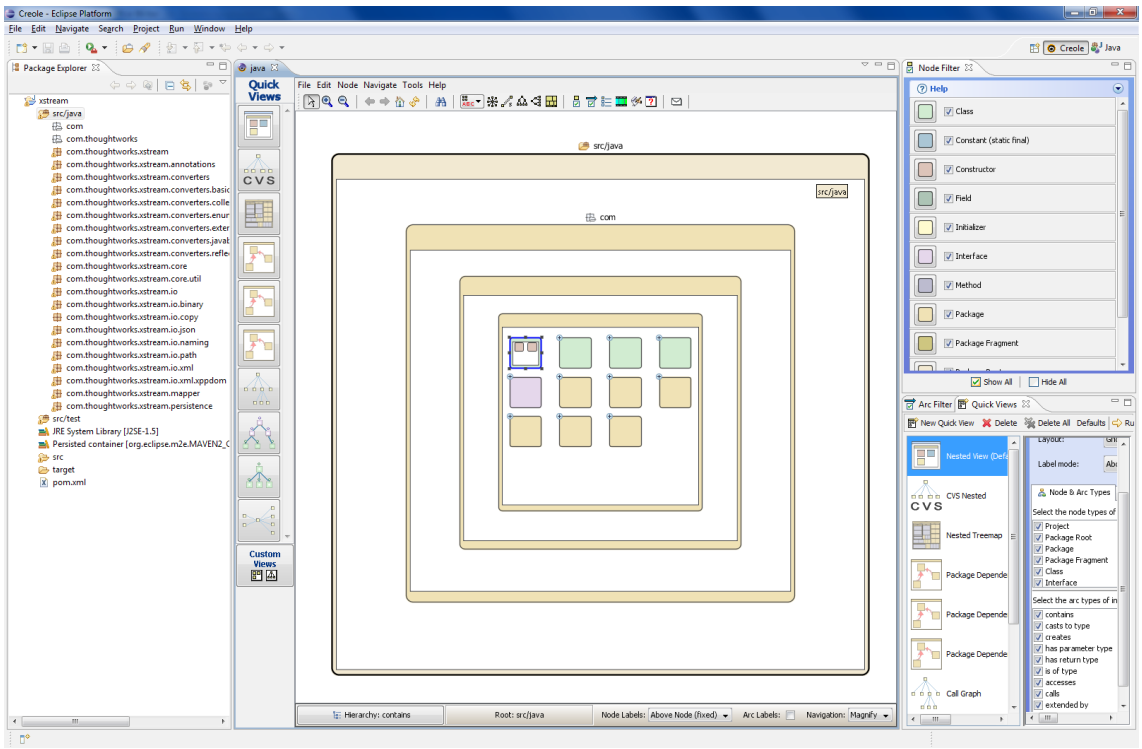

Abbildung 23: Target-Software XStream, angezeigt in Creole

Creole bietet eine Reihe an voreingestellte Views an [Chisel11]:

- Nested View
- CVS Nested
- Nested Treemap
- Package Dependencies
- Call Graph
- Class Hierarchy
- Fan In/Out

Zusätzlich ist es möglich, eigene Views zu kreieren.

In Creole besteht jedes Diagramm aus Nodes und Verbindungen. Neben vorgefertigten Ansichten, ist es auch möglich selbst festzulegen, was angezeigt wird. Nodes entsprechen dabei Teilen von einem anderen Node. Zum Beispiel besteht der Package-Node aus anderen Package-/ und Class-Nodes. Verbindungen stellen hierbei Relationen zwischen den einzelnen Source-Teilen dar. Eine Ableitung wäre so eine Relation.

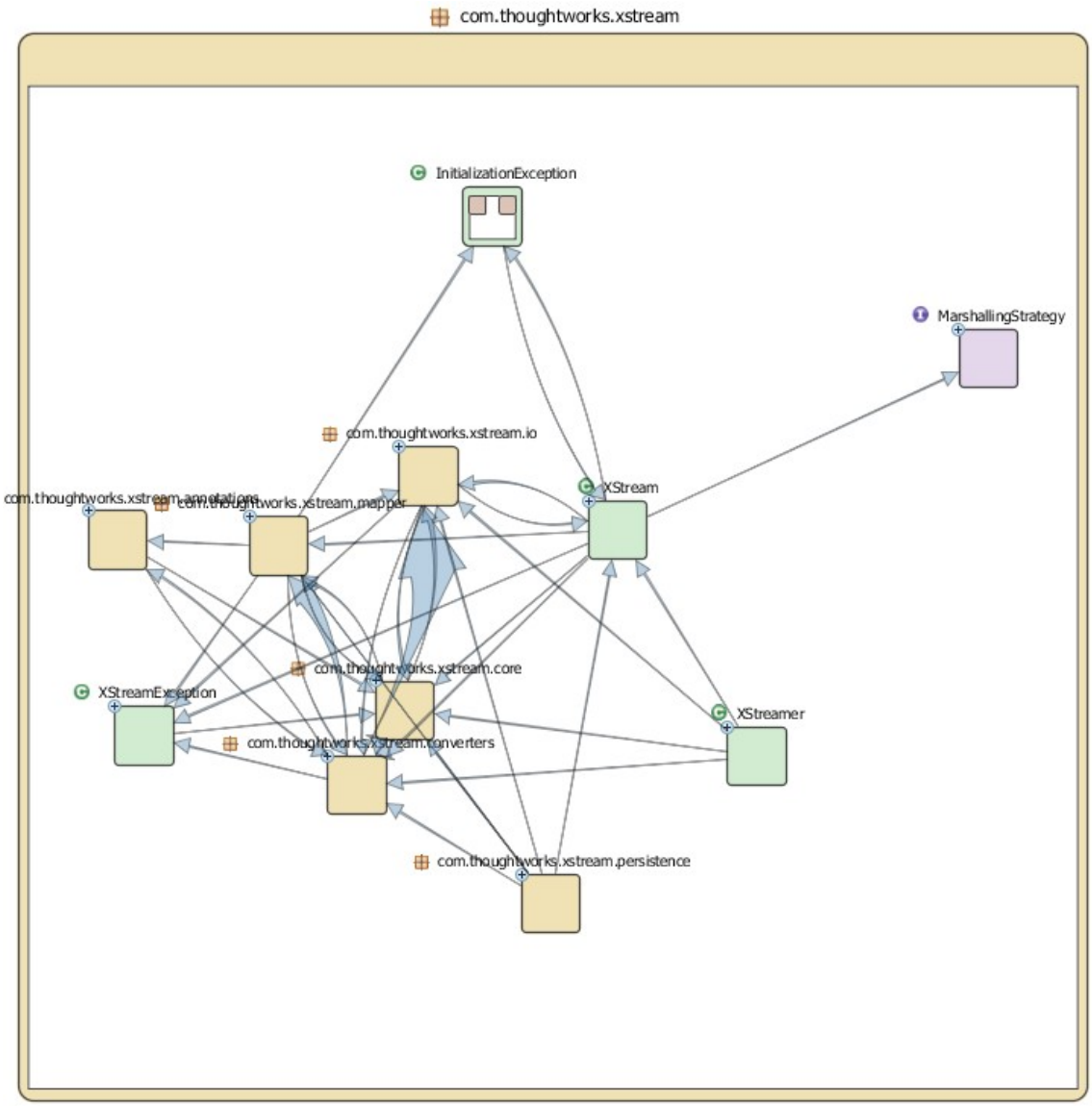

Abbildung 24: Dependency Diagram in Creole

Die Visualisierung erinnert an das in 2010 vorgestellte Konzept der Code Bubbles [BRZKCKCAL10]. Bei jedem Node welcher Quellcode beinhaltet, ist es möglich, diesen anzuzeigen. Anzumerken ist, dass Creole älter ist.

Creole ist die Einbettung von ShriMP in Eclipse. Die aktuelle Version ist aus dem Jahr 2009 und wurde seither auch nicht mit offiziellen Bug-Fixes gewartet.

4.4.2.1 Evaluationsergebnis Creole

Creole läuft nur in der Eclipse-Version 3.3. Eine Portierung auf eine höhere Version war nicht ohne großen Mehraufwand möglich. Weiters hat die aktuelle Version des Eclipse-

Plugins Probleme mit der Maussteuerung. Es ist zu empfehlen, SHriMP zu verwenden.

Zu Beginn wirkt die Präsentation der Daten überragend gut, aber ein wirklich produktives und schnelles Navigieren innerhalb der Architektur ist nicht möglich.

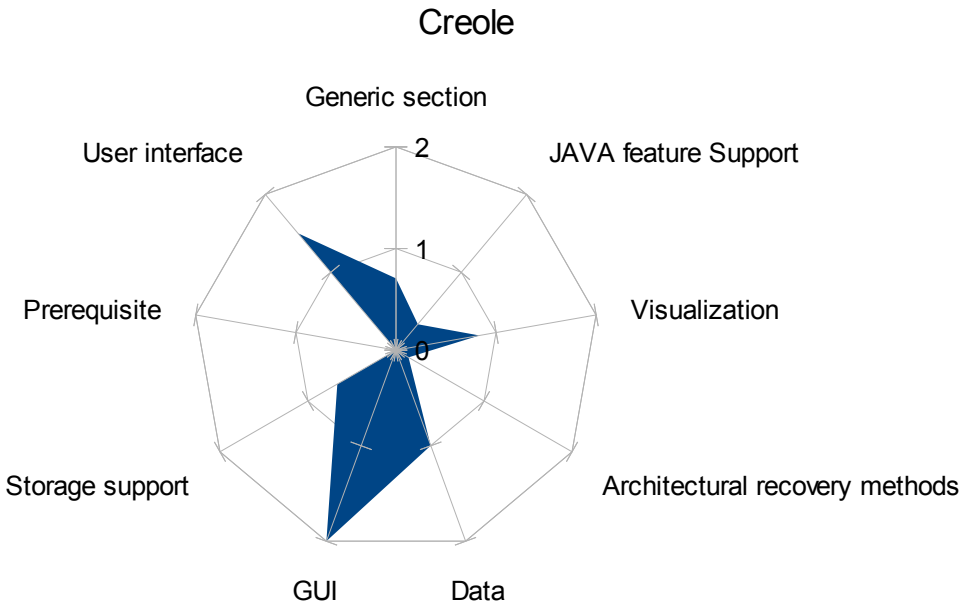

Abbildung 25: Analyse-Ergebnis Creole

In Abbildung 25 sieht man das Evaluations-Ergebnis im Allgemeinen. Creole ist eher eine Visualisierung von Source-Code als eine Analyse-Software an sich.

4.4.3 jRMTool

Das *jRMTool* [JRM2011] ist eine Implementierung der AR-Methode *Reflexion Models*. Es ist als Eclipse-Plugin verfügbar und ermöglicht die Anwendung der Methode auf Java-Code.

Auf Basis der Architektur-Beschreibung von XStream[7] und der Package-Aufteilung des Projekts wurden ein *High-Level-Model* und eine *Mapping-Datei* erstellt.

7 http://xstream.codehaus.org/architecture.html

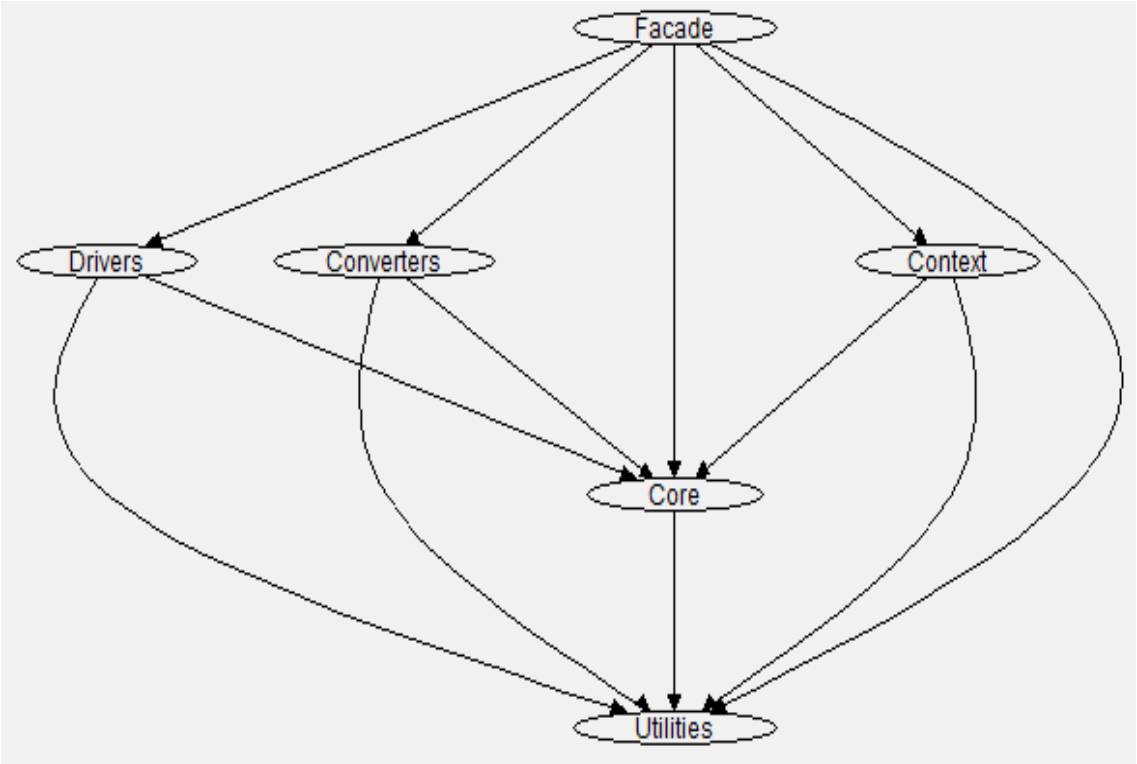

Abbildung 26: High-Level-Model von XStream

Die Mappings sind in XML-Form abzubilden.

```
<map>
<entry package="^com.thoughtworks.xstream.converters"
mapTo="Converters"/>
<entry package="^com.thoughtworks.xstream.io" mapTo="Drivers"/>
<entry class="^XStream$" mapTo="Facade"/>
<entry class="Marshall" mapTo="Context" />
<entry package="^com.thoughtworks.xstream.core$" mapTo="Core" />
<entry package="^com.thoughtworks.xstream.core.util$" mapTo="Utility"
/>
</map>
```

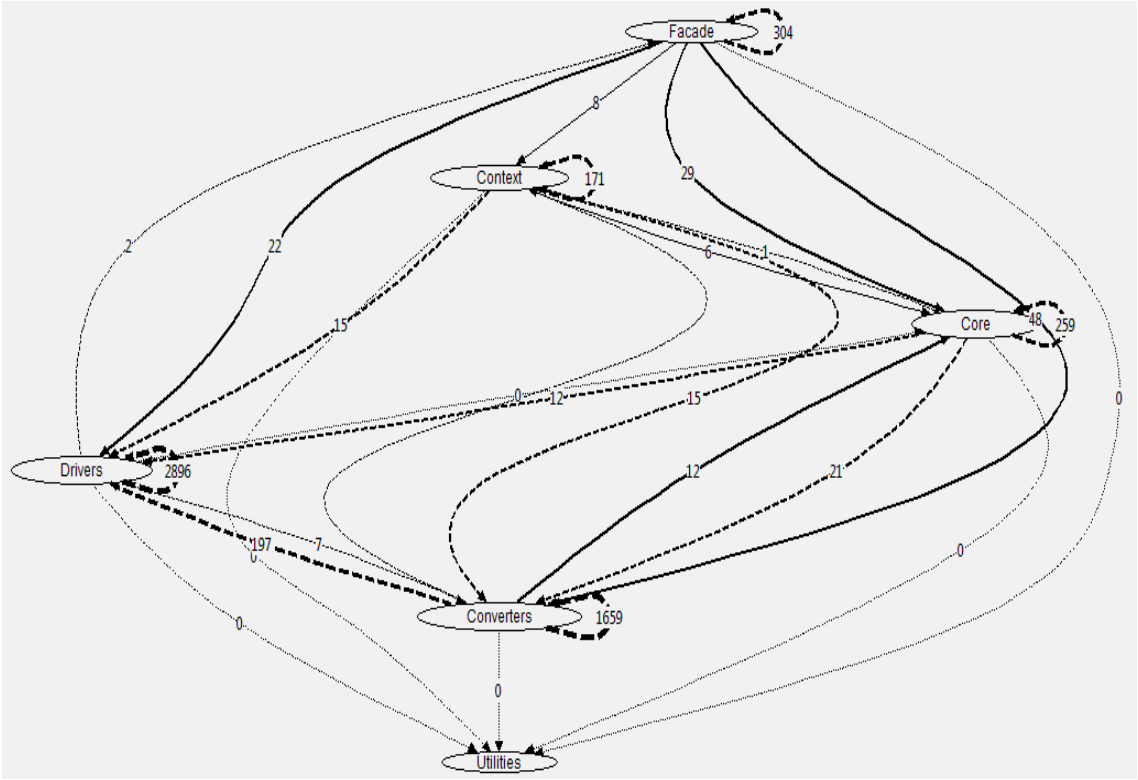

Abbildung 27: Errechnetes Software Reflexion Model von XStream

Es ist in Abbildung 27 sehr gut ersichtlich, dass es eigentlich keine Verbindung zwischen Core und Utilities gibt. Aufgrund ihrere Packages wäre jedoch eine Abhängigkeit zu erwarten.

Das jRMTool wird zwar nicht mehr gewartet, bietet sich aber an, zu Beginn eines AR-Projekts eingesetzt zu werden. Es ist sehr schnell möglich, Architekturaussagen aus Interviews zu überprüfen.

4.4.3.1 Evaluationsergebnis jRMTool

Wenn es keine Initialinformationen über ein Projekt gibt, ist es natürlich nicht möglich ein Model und ein Mapping zu erarbeiten. Automatisiert wird eigentlich nur die Erstellung des *Reflexion Models*. Bei genauerer Betrachtung dieser Methode, wird ersichtlich, dass dies tatsächlich so gewollt ist.

Die Dokumentation des Eclipse-Plugins ist sehr spartanisch, reicht aber aus, sobald sie gefunden wird.

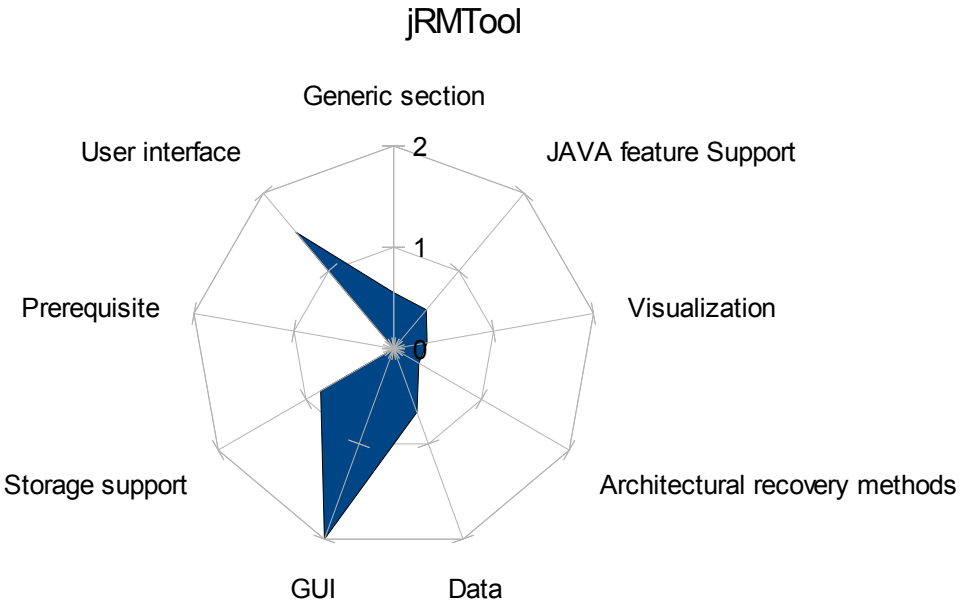

Abbildung 28: Analyse-Ergebnis jRMTool

Das Diagramm in Abbildung 28 zeigt das Analyse-Ergebnis. Als einzige AR-Methode wird, wenig überraschend, Reflexion Models unterstützt. Leider gibt es keine Unterstützung für weitere Java-Features wie Generics oder Annotations. Theoretisch lassen sich diese Erweiterungen aber selbst hinzufügen. Der Ersteller des Tools hat es seit 2009 nicht mehr weiterentwickelt.

4.4.4 X-Ray

X-Ray ist die Weiterentwicklung von *CodeCrawler* als Eclipse-Plugin. Ziel ist es, Software innerhalb von Eclipse zu visualisieren. Dafür werden Polymetric Views verwendet. Die eingesetzten Metriken sind in Abbildung 29 ersichtlich.

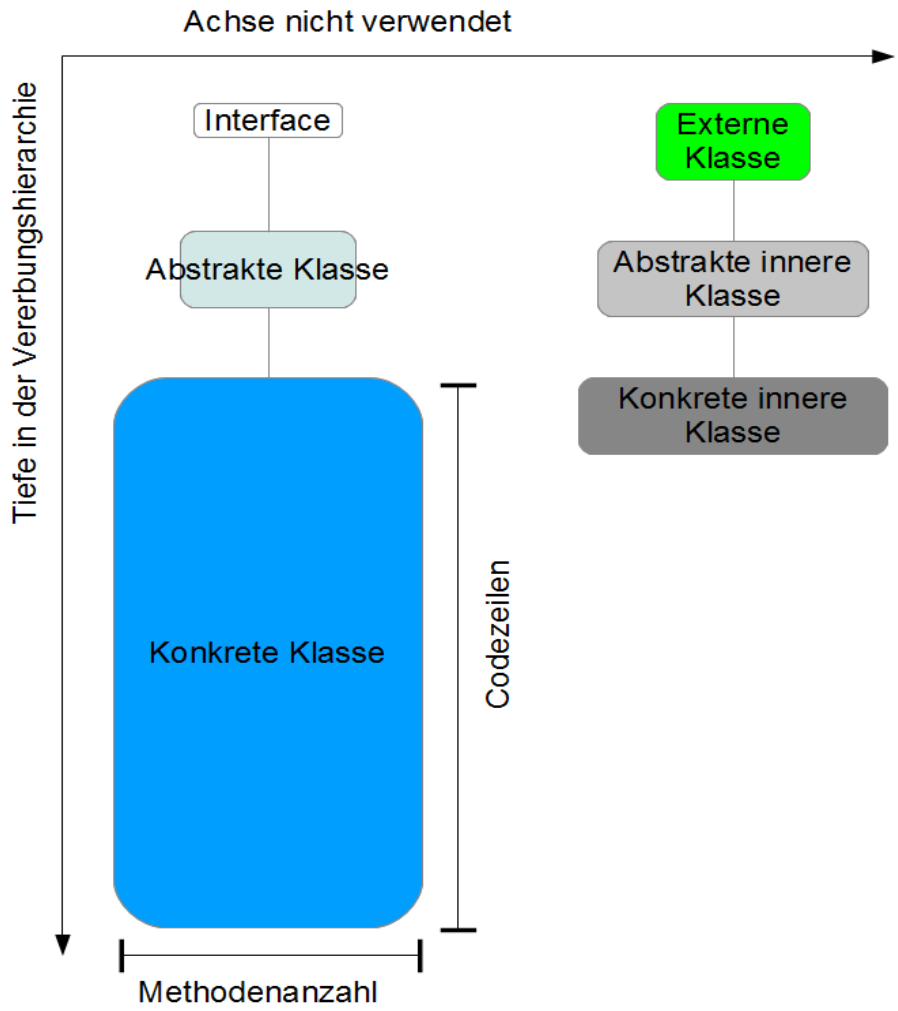

Abbildung 29: Verwendete Metriken im Polymetric View von X-Ray

Für das dahinterliegende Datenmodell sind bereits Addons (wie die Visualisierung als Stadt) vorhanden. Diese werden aber nicht untersucht, da sie zum Zeitpunkt der Evaluation nicht verfügbar waren.

4.4.4.1 Evaluationsergebnis X-Ray

X-Ray ist sehr spezialisiert auf die Anzeige seiner vorgefertigten *Polymetric views* und *Abhängigkeitsdiagramme*. Seine einfache Bedienung ermöglicht es, dieses Tool zu Beginn eines Projekts einzusetzen, um eine Gesamtübersicht zu erhalten. Außerdem kam es beim Einsatz des Code-Parsers bei keinem Projekt zu Fehlermeldungen oder unvorhergesehenem Verhalten.

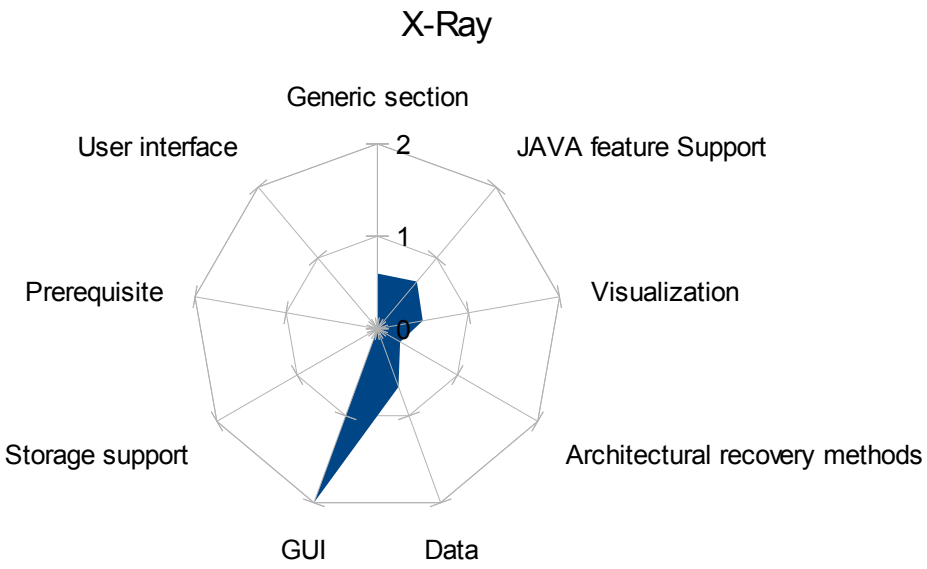

Abbildung 30: Analyse-Ergebnis X-Ray

Das Analyseergebnis unterstreicht den ersten Eindruck. X-Ray ist kein Tool für den gesamten AR-Prozess, sondern bietet sich an, eine schnelle Überprüfung von Interview-Ergebnissen zu machen. Eventuell ist es auch sinnvoll, es während eines Interviews einzusetzen, da die Bedienung einfach und wenig verwirrend ist.

4.4.5 Moose Tool Suite

Das *Moose Tool Suite* ist eine in *Smalltalk* entwickelte Software-Analyse-Plattform. Aus bestehender Software wird ein *FAMIX-Modell* extrahiert, dass die Struktur der Software technologieunabhängig speichert. Dieses FAMIX-Modell wird als Input für die Moose Tool Suite verwendet. Zur Extrahierung gibt es kommerzielle Applikationen (*inFusion*) und Open-Source-Tools (*verveineJ*). Für die Beschreibung von dynamischen Zusammenhängen steht das *DYNAMIX-Modell* zur Verfügung.

Der Workflow in Moose ist in drei Teile aufgeteilt. In Abbildung 31 ist eine schematische Darstellung zu sehen. *Importers* kümmern sich um die Übersetzung der spezifischen Programmiersprache in das technologieunabhängige Modell (in der Regel FAMIX). Auf Basis der *models* ist es möglich, diverse Analysen anzuwenden. Diese drei Workflowschritte sind erweiterbar. So ist es möglich eigene Code-Parser zu schreiben, um beispielsweise eine Delphi-Anbindung zu erzeugen.

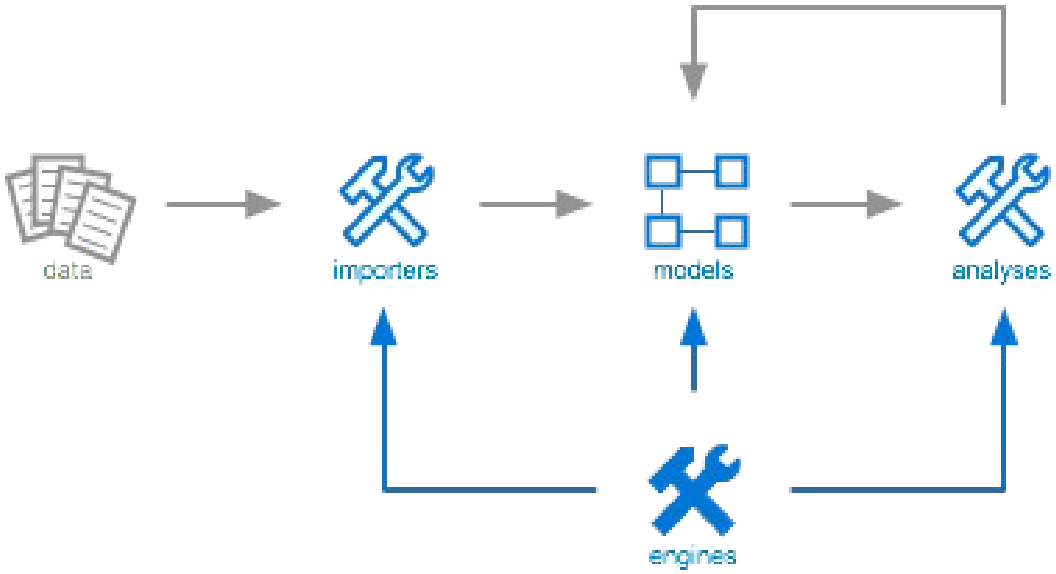

Abbildung 31: Workflow im Moose Tool Suite
Quelle: [Gir11]

Im Rahmen der Evaluation wurden auch folgende Erweiterungen für Moose untersucht:

- *inFusion/inCod*e (Code-Smells & Overview-Matrix)
- *CodeCity* (Visualisierung der Code-Struktur als Stadt)
- *Kumpel* (Visualisierung der Code-Evolution)

4.4.5.1 Evaluationsergebnis Moose Tool Suite

Moose hat durch seine Nicht-Integration in eine (oft genutzte) Entwicklungsumgebung den Vorteil, dass keine Inkompatabilitäten zwischen verschiedenen IDE-Versionen auftreten. Tools, die zum Beispiel eine Integration in Eclipse haben, sind oft mit aktuellen Eclipse-Versionen inkompatibel.

Das *Moose Book* [Gir11] erleichtert einen Einstieg in Moose. Leider ist es aber nicht mehr aktuell und die Dokumentation zu den einzelnen Plugins teilweise veraltet.

Als Technologie für die Entwicklung von Moose wurde Smalltalk gewählt. Der Vorteil dieser Sprache ist die Unterstützung sehr vieler Paradigmen der objektorientierten Programmierung. Im nicht-akademischen Bereich ist die Verbreitung aber sehr gering (siehe [GitHub11] und [LaIn11]). Deshalb fällt es schwer, ad-hoc neue Addons zu entwickeln oder bestehende zu warten. Eine solche Herausforderung gab es während der

Evaluation beim Verwenden von Kumpel.

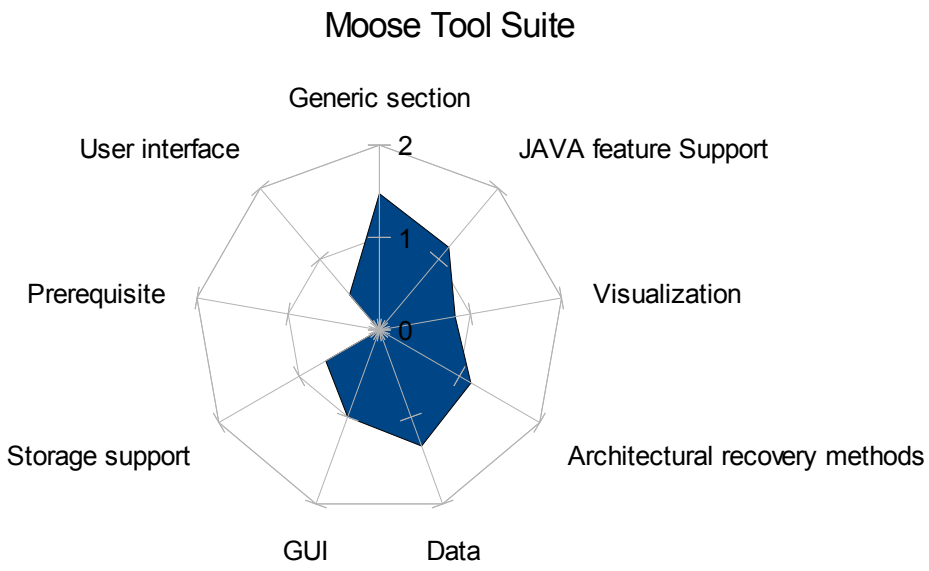

Abbildung 32: Analyse-Ergebnis Moose Tool Suite

Auf dem Diagramm in Abbildung 32 ist sehr gut ersichtlich, dass die Moose Tool Suite ein Framework für das Architectural Recovery ist. Durch den starken Fokus auf Erweiterbarkeit und Entkopplung der Komponenten bieten sich einige Möglichkeiten zur Customisierung. Außerdem sind Bibliotheken vorhanden, die Scripting und Visualisierung erleichtern. Die Software ist auch in Industrieprojekten erprobt. Des Weiteren ist die Moose Tool Suite auch eines der wenigen Programme, das die Abhängigkeiten bei *Annotations* erkennt.

4.4.6 Fujaba4Eclipse

Fujaba ist ein eigenständiges Programm, welches Round-Trip-Engineering ermöglichen soll. Ein Teil des Round-Trip-Engineering ist das Reverse Engineering. Fujaba bietet für diese Möglichkeit *Reclipse* an. Dies ist ein Addon für das Eclipse-Plugin *Fujaba4Eclipse* (eine Portierung auf die Eclipse-Architektur). Mittels Reclipse ist es möglich, aus einer bestehenden Java-Code-Basis ein Model der Software zu extrahieren. Beim Reverse Engineering können Klassen- und Aktivitätsdiagramme automatisch generiert werden. Die Qualität dieser Diagramme ist aufgrund der fehlenden Semantik häufig wenig zufriedenstellend. Metriken können ebenfalls berechnet und aufgesplittet auf die einzelnen Modellkomponenten angezeigt werden.

Das Herzstück von Reclipse ist aber der *Design-Pattern-Detection-Mechanismus*. Patterns können frei generiert oder mit etwas Aufwand importiert werden. Es ist möglich eine statische Analyse auf Basis des Modells durchzuführen. Ergänzend kann eine dynamische Analyse gemacht werden. Dadurch ist es möglich, auch Verhaltens-Entwurfsmuster zu erkennen oder bereits definierte strukturelle Entwurfsmuster zu verifizieren.

4.4.6.1 Evaluationsergebnis Fujaba4Eclipse

Es ist allerdings sehr schwer, dieses Programm einsatzfähig zu bekommen, da es stark an aktueller Dokumentation mangelt. Auf der Fujaba-Homepage werden mehrere unterschiedliche Wege der Installation von Reclipse angeführt. Keine davon funktioniert zufriedenstellend. Der Source-Code wird auch nicht automatisch mit dem Eclipse-Plugins ausgeliefert. Eine Erweiterung oder ein Bug-fixen ist dadurch nur über den Umweg des Fujaba-SVN-Servers möglich.

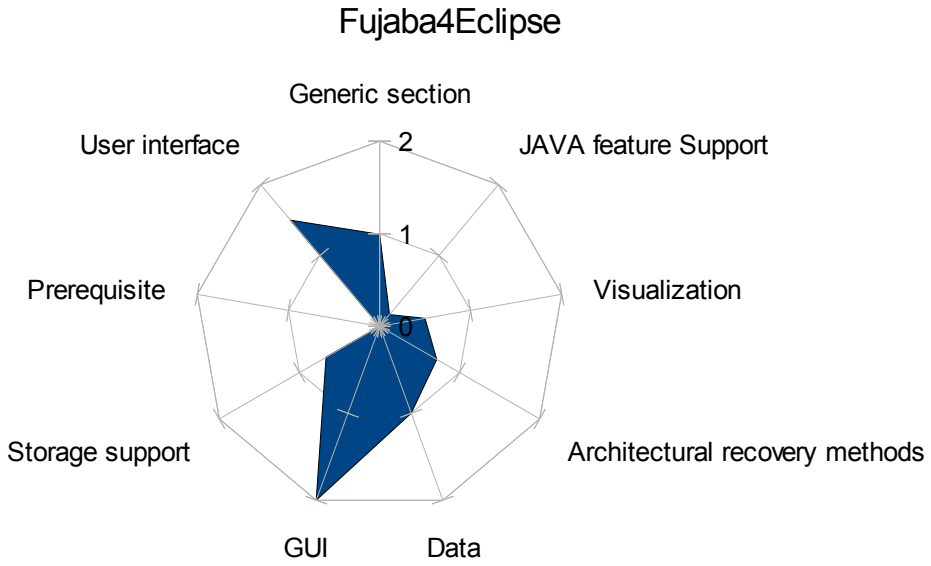

Abbildung 33: Analyse-Ergebnis Fujaba4Eclipse

Der Fokus in Reclipse liegt eindeutig auf der AR-Methode *Design Pattern Detection*. Da der Code-Parser nur Java 1.4 als Input akzeptiert, war es nicht möglich Generics und Annotations zu analysieren.

4.4.7 Q-Impress

Q-Impress ist ein junges Tool-Suite, welches den gesamten Entwicklungsprozess begleiten und dadurch die Qualität der Software verbessern soll. Für die Evaluation ist nur die Reverse Engineering-Komponente *SoMoX* interessant. Der *SoftwareModelXtractor* extrahiert Komponenten aus dem Source-Code. Die daraus entstehenden Modelle sollen helfen, die Software zu verstehen. Aus den Daten des Source-Codes lässt sich ein GAST-Modell erstellen [BHTKK10]. Auf Basis dieses GAST-Modells wird das restliche Round-Trip-Engineering aufgebaut.

4.4.7.1 Evaluationsergebnis Q-Impress

Die aktuelle Release-Version 1.0.0.2 von Q-Impress beinhaltet noch kein SoMoX. Deshalb musste das Nightly-Build verwendet werden. Der Code-Parser ist nicht ausführbar, wenn das Dependency-Management des Eclipse-Projektes von einem anderem Nicht-Standard-Plugin wie *Maven Integration for Eclipse* durchgeführt wird.

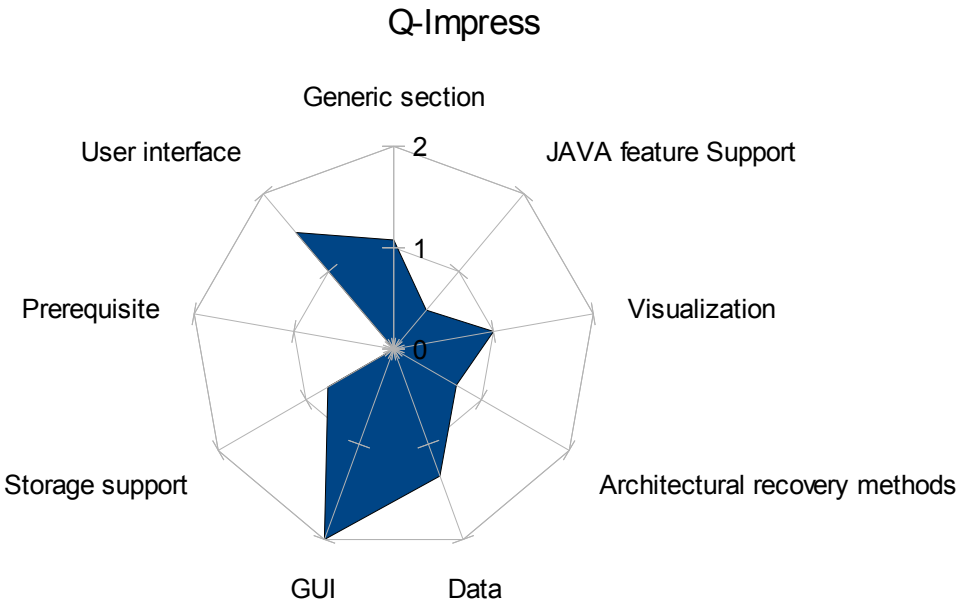

Abbildung 34: Analyse-Ergebnise Q-Impress

Dank der Einbettung in Q-Impress ist es möglich, auch *Klassen- und Komponentendiagramme* zur Visualisierung zu verwenden. Weiters können *Call-Graphen* angezeigt werden.

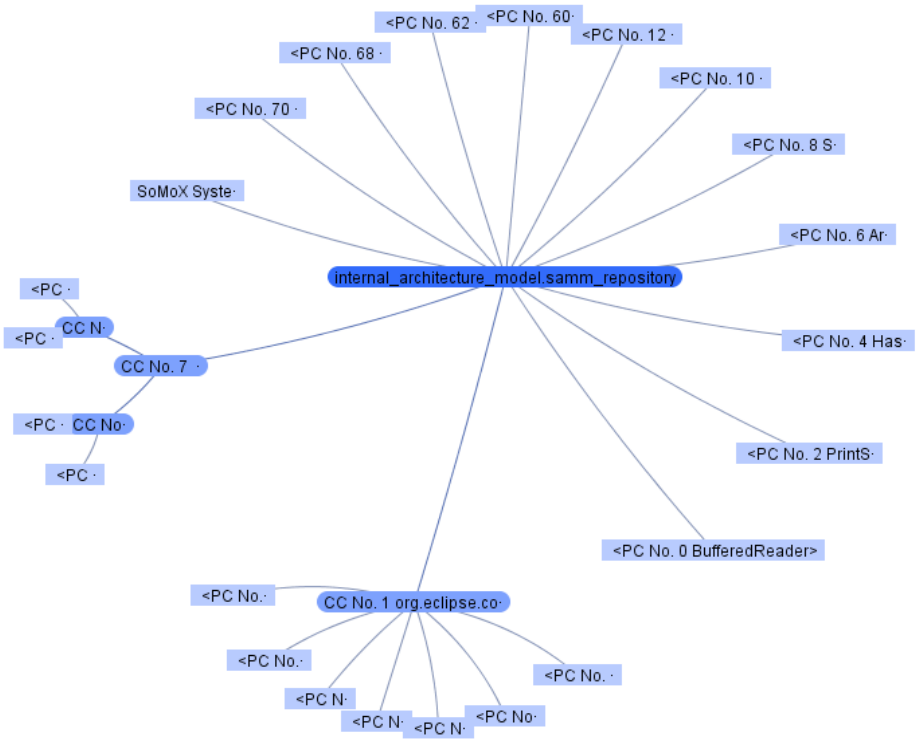

Abbildung 35: org.eclipse.compare.core dargestellt als hyperbolischer Baum in SoMoX

SoMoX fügt noch hyperbolische Bäume (siehe Abbildung 35) und eine Rectangular Map hinzu. Für Letzteres ist kein wirklicher Nutzen erkennbar.

5 Barrieren der aktuellen AR-Landschaft

5.1 Organisatorische Rahmenbedingungen

Interessanterweise scheinen die Grenzen beim semi-automatisierten AR teilweise auch organisatorischer Natur zu sein. Die eingesetzten Tools etwa, die den AR-Prozess unterstützen sollen, werden teilweise nicht mehr gewartet.

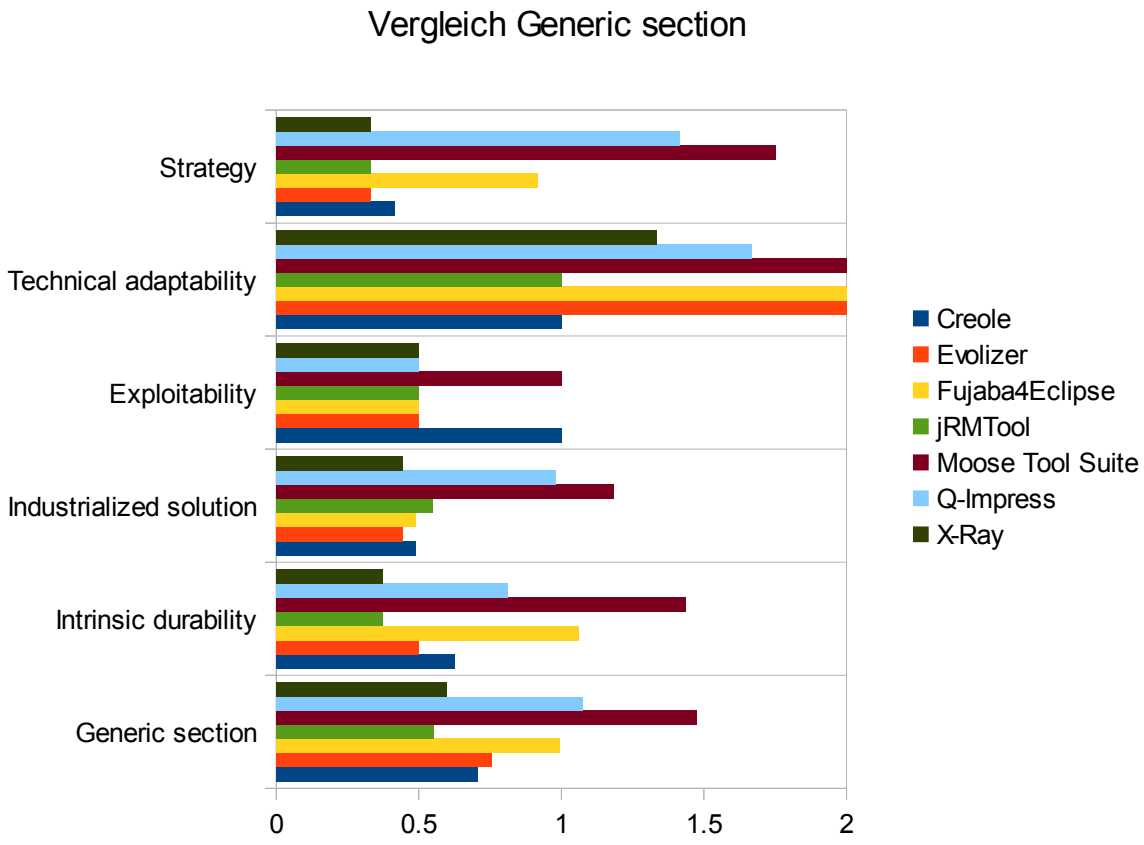

Abbildung 36: Vergleich des Projektumfelds bei AR-Tools

Die *Generic section* des Evaluations-Sheets befaßt sich mit dem Projektumfeld und generellen Wartungsfragen. Wenn ein Tool eingesetzt wird, erhofft man sich ein möglichst stabiles Produkt. Da die AR-Applikationen selbst Expertenenwendungen sind, ist meist eine Einarbeitung sehr schwierig und zeitaufwändig (siehe Abbildung 36, Punkt *Exploitability*). Deswegen ist es natürlich von Interesse, das gewonnene Know-How weiterzuverwenden.

Viele der getesteten AR-Anwendungen haben keinen öffentlichen Release-Plan oder ein aktives Support-System (*Strategy und Intrinsic durability*). Aufgrund der Zeitsensitivität eines AR-Projekts [DeDuNi08], ist es wichtig, dass eine aktuelle Dokumentation

vorliegt (*Industrialized solution*).

Manchmal reicht der mitgelieferte Umfang des unterstützenden Tools nicht aus, um die Code-Base zu analysieren. In diesem Fall ist es wichtig, dass dieses erweitert werden kann (*Technical adaptability*). Hier ist es möglich, dass es eine Grenze beim Know-How der verfügbaren Mitarbeiter im Projektteam gibt. Die Moose Tool Suite ist beispielsweise in Smalltalk implementiert, welches im Industriesektor kaum Verbreitung findet [GitHub11] [LaIn11].

Eine Limitation für das Durchführen eines AR-Projektes können also auch die Umgebungsfaktoren der eingesetzten Tools sein.

5.2 Usability

Semi-automatisierte AR-Methoden müssen natürlich von Applikationen implementiert werden. Da diese Input vom User erwarten und nicht autark agieren, muss ein User-Interface vorhanden sein. Die Zugänglichkeit dieses User-Interfaces kann sich ebenfalls als Grenze beim Einsatz von semi-automatisierten AR manifestieren.

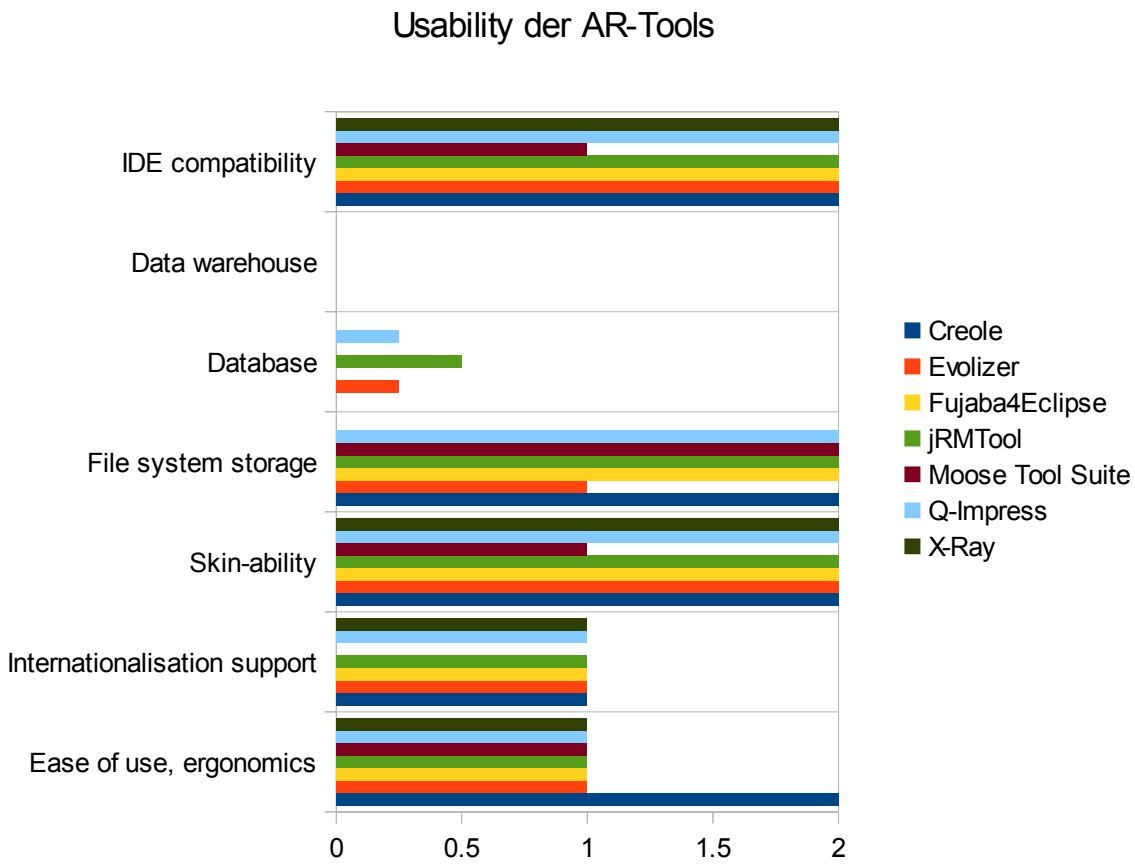

Abbildung 37: Vergleich der Usability der AR-Tools

In Abbildung 37 kann ein Vergleich der Criterions gesehen werden, welche sich mit dem User-Interface (*Internationalisation support, IDE compatability, Skin-ability*) und der Usability der Software allgemein befasst (*Ease of use, ergonomics*). Gerade die Einbettung in eine IDE (*IDE compatability*) kann aber auch Nachteile haben. Viele der erfassten AR-Tools sind Plugins für Eclipse. Da es zwischen unterschiedlichen Eclipse-Versionen aber auch zu Änderungen im Eclipse-Framework kommen kann, besteht die Gefahr, dass das AR-Tool nicht mit einer aktuellen Eclipse-Version kompatibel ist.

Zusätzlich ist es auch wichtig, die gewonnenen Daten zu speichern. Dies kann auf unterschiedlichen Wegen erfolgen (*File system storage, Database*). Sollten viele AR-Projekte durchgeführt werden, ist es wertvoll, diese im Gesamten zu analysieren (*Data warehouse*), um eventuell vorhandene Korrelationen herauszufinden.

5.3 Visuelle Grenzen

Bei der Analyse des Systems sollen die gewonnenen Daten möglichst gut für den Menschen begreifbar aufbereitet werden. Aus großen Applikationen werden große Datenbasen über den Source-Code gesammelt. Hier fällt es schwer, automatisiert dem User semantisch zusammenhängende Komponenten zu präsentieren. Die fehlende Semantik lässt sich auch nicht ohne User-Eingaben der Analyse hinzufügen. Es gibt Ansätze, gewisse Komponenten auf Basis ihrer Attribute (z.b. Metriken) zu kategorisieren [LaRa06]. Da Software von Menschen gebaut wird, kann eine 100 %ige Trefferquote aber nicht vorkommen.

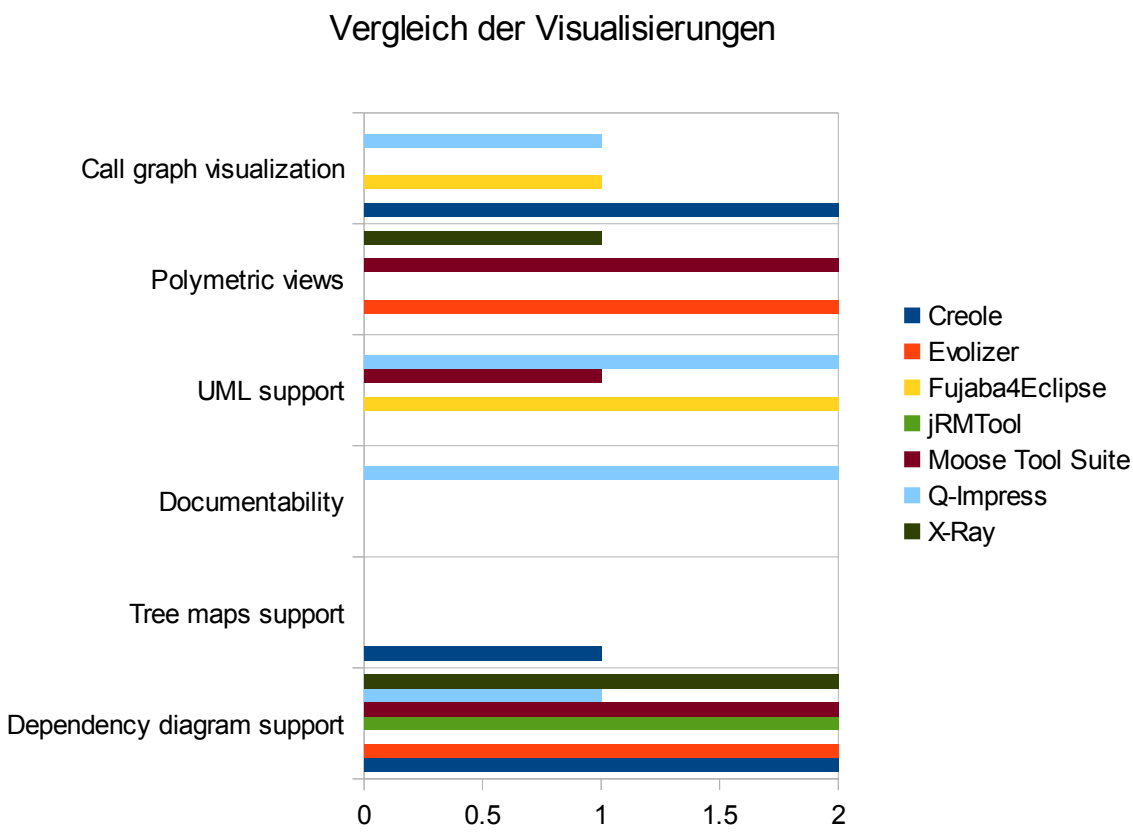

Abbildung 38: Vergleich der Visualisierungsmöglichkeiten in den einzelnen Tools

In Abbildung 38 ist eine Übersicht über die unterstützten Visualisierungen zu sehen. Auffallend hierbei ist, dass die Möglichkeit der Tree-Maps (*Tree maps support*) scheinbar nur sehr rudimentär verwendet wird. Vor allem *Polymetric views* sind sehr oft in irgendeiner Form in den AR-Tools vorhanden. Die Möglichkeit, direkt im Diagramm Kommentare zu hinterlassen (*Documentability*), um Analyse-Ergebnisse sofort

festzuhalten, ist kaum unterstützt.

Die Visualisierungen sind oft sehr technisch und detailliert. Für Dokumentationszwecke und Präsentationen ist es deswegen von Vorteil, die Darstellung zu vereinfachen. Ein Verstehen fällt leichter, wenn Metaphern verwendet werden. CodeCity – eine Visualisierung auf Basis der Moose Tool Suite – geht diesen Weg und stellt eine Architektur als Stadt dar (Abbildung 39).

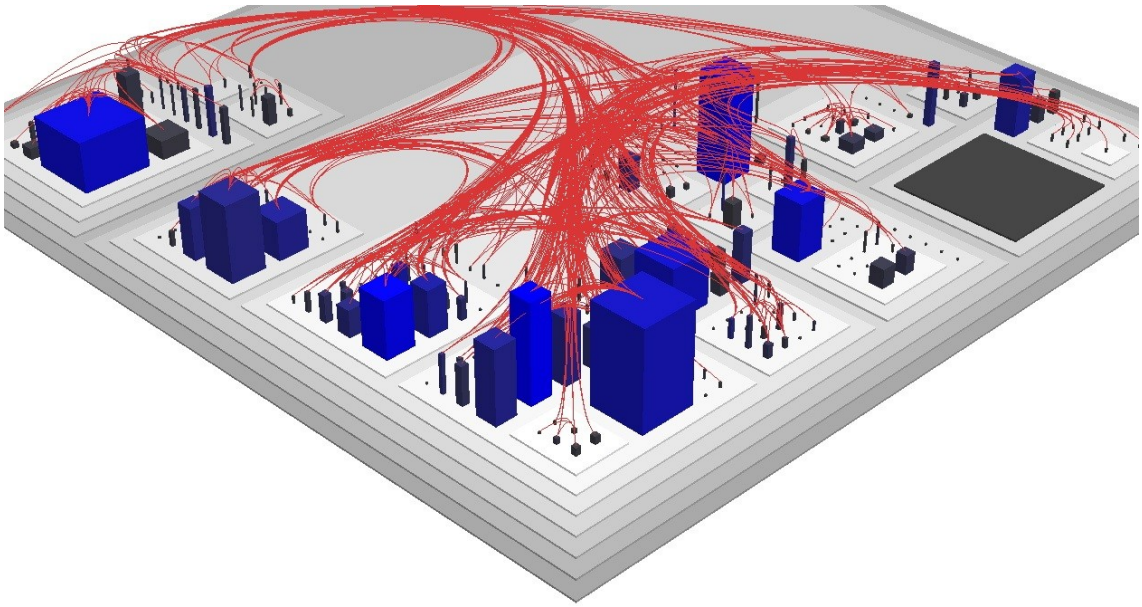

Abbildung 39: Visualisierung einer Architektur in CodeCity

Obwohl die Ansicht eigentlich nichts anderes als ein Polymetric view im 3D-Raum ist, fällt das Begreifen - dank der Metapher Stadt - um einiges leichter. Zusätzlich kann aus extrahierten Evolutions-Daten das Bauen der „Stadt“ beobachtet werden.

5.4 Grenzen der Code-Analyse

Bis auf *Concern graphs* wird jede aufgezeigte AR-Methode von einem Tool implementiert (siehe Abbildung 40). Es gibt zwar AR-Applikationen, die das Konzept der *Concern graphs* unterstützen, diese sind aber nicht mehr funktional. Wenn diese Methode eingesetzt werden soll – was vor allem beim Reengineering sehr sinnvoll ist – musst dafür erst wieder eine funktionierende Tool-Unterstützung hergestellt werden.

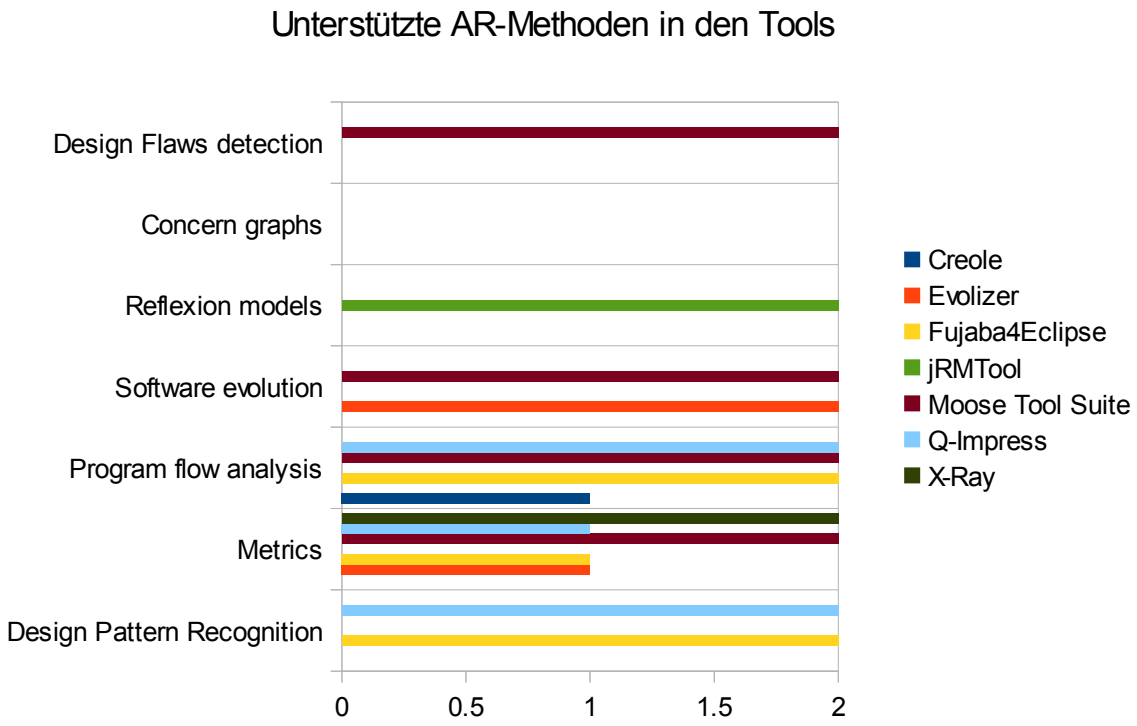

Abbildung 40: Vergleich der Tool-Unterstützung der AR-Methoden

Metriken werden beinahe immer berechnet, aber oft nicht dem User präsentiert.

Eine Überprüfung der unterstützten, erweiterten Sprach-Features von Java 1.6 hat interessante Ergebnisse geliefert (Abbildung 41). Die Abhängigkeiten, eingeführt durch *Generics,* wurde von nicht einmal der Hälfte der untersuchten AR-Applikationen unterstützt. Noch ernüchternder war das Ergebnis bei den *Annotations*. Einzig die Moose Tool Suite hat die Abhängigkeit korrekt erkannt.

Gerade im Architectural Recovery ist es interessant, die vorhandenen Code-Kommentare ebenfalls in die Analyse einfließen zu lassen. Keiner der AR-Tools präsentiert aber vorhandene Code-Kommentare (*JavaDocs*) oder beziehen sie in die Analyse mit ein. Die Unterstützung von JavaDocs, Annotations und Generics ist auf jeden Fall eine derzeitige Barriere die aufgelöst werden könnte.

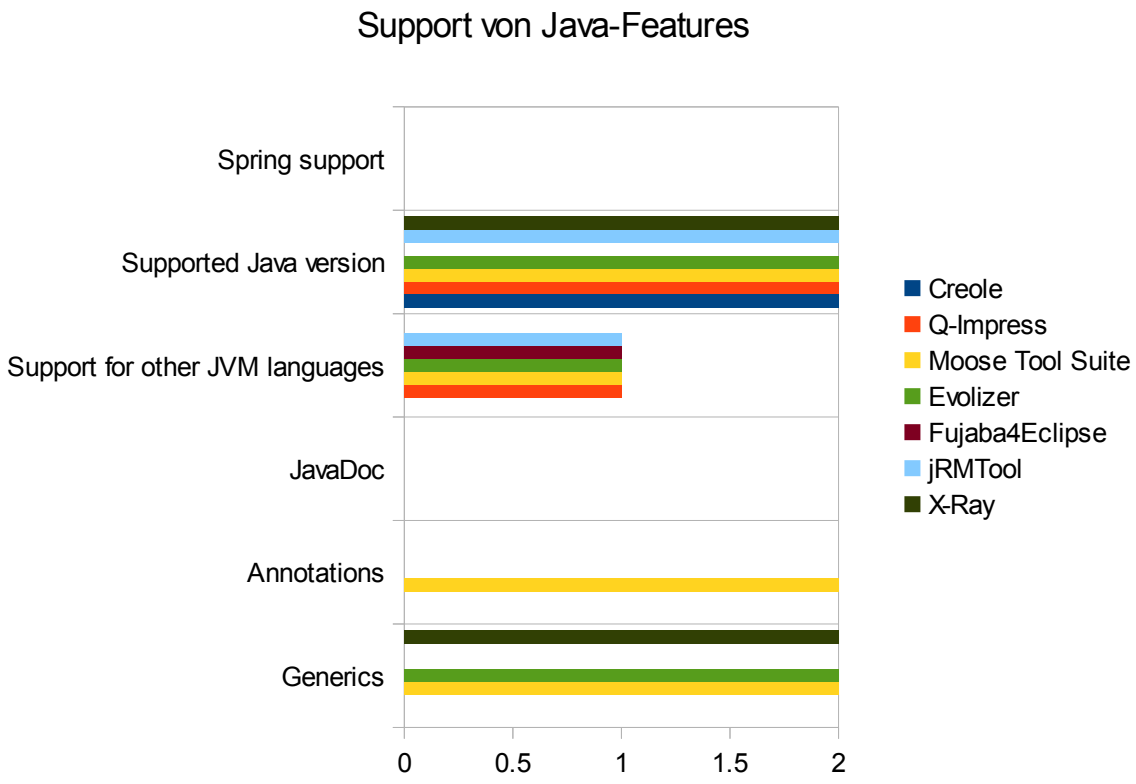

Abbildung 41: Support von erweiterten Sprach-Features von Java 1.6

Die Unterstützung für andere JVM-Sprachen wie Scala[8] ließe sich auf jeden Fall zu AR-Applikationen hinzufügen, sollten diese die Code-Analysen auf Basis eines eigenen Datenmodells machen. Die Umwandlung des Source-Codes in ein technologieunabhängiges Datenmodell bringt aber auch einen Nachteil: Bei einer korrekten, technologieagnostischen Definition des Datenmodells, fällt es schwer von Entitäten im Modell auf die Realisierung im Source-Code zurückzuspringen. Dies ist bei der Analyse eines Systems jedoch ein wichtiger Schritt [DeDuNi08].

Die Unterstützung eines Dependency-Injection-Frameworks wie Spring, bietet eine komplett neue Herausforderung für einen Code-Analyzer. Grund dafür ist, dass diese Frameworks Abhängigkeiten auflösen sollen. Dadurch entstehen Abhängigkeiten, die in Konfigurationsdateien ausgelagert sind. Code-Analyzer müssten diese also auch mit untersuchen.

Eine andere Möglichkeit ist natürlich die dynamische Code-Analyse (Analyse zur Laufzeit), die derzeit nur von Q-Impress unterstützt wird. Dadurch werden die real

8 Website von Scala: http://www.scala-lang.org/

vorhandenen Abhängigkeiten sehr gut aufgelöst. Hier kann es aber folgende Fragen geben, deren Antworten eine nicht überwindbare Analyse-Grenze sein können:

- Kann der Tracer/Observer in die Host-Applikation eingeschleust werden?
- Ist das System obfuskiert?
- Haben die Analyse-Tätigkeiten Einfluss auf das System [Schr35]?

Wenn es Abhängigkeiten zu externen Bibliotheken gibt, stellt sich die Frage, ob diese auch aufgenommen werden sollen. Bei der Kompilierung und Ausführung von Java-Applikationen muss der Suchpfad für diese Bibliotheken übergeben werden (Classpath). Dieser Classpath wird durch Build-Tools wie Maven[9] zum Verwendungszeitpunkt automatisch hinzugefügt. Das Dependency-Management erfolgt also auch dort. Derzeit unterstützt noch kein untersuchtes AR-Tool Maven.

9 Website von Maven: http://maven.apache.org/

6 Fazit

Durch Software-Evolution wird das Delta zwischen der vorhandenen Dokumentation und der tatsächlichen Implementierung immer größer. Dieses Delta zu verringern ist die Aufgabe eines AR-Projekts. Klassische Methoden, wie Interviews, werden mit semi-automatisierten Methoden kombiniert, um einen möglichst effizienten Ressourceneinsatz zu gewährleisten.

Eine Toolunterstützung für diese Methoden ist Großteils bereits vorhanden. Die Qualität dieser Anwendungen hat eine sehr starke Streuung. Oft bedürfen diese noch selbst Implementierungsaufwand, um für das Projekt einsatzfähig zu sein. Erfreulicherweise ist die Usability der untersuchten Anwendungen zufriedenstellend. Leider gibt es kaum verwendbare und aktuelle Dokumentation.

Die Menge der Informationen, welche durch eine AR-Methode zusammengetragen werden, müssen für den Menschen aufbereitet und präsentiert werden. Viele Tools setzen hierbei *Polymetric views* ein. Mittels Metaphern ist es möglich, diese technischen Ansichten noch besser begreifbar zu machen.

Erstaunlicherweise scheitern nahezu alle Code-Analyse-Methoden an der korrekten Kategorisierung von erweiterten Java-Features, die bereits seit 2004[10] vorhanden sind. Dass es im Bereich *Dependency-Injection* Barrieren gibt, war wenig überraschend. Diese Frameworks sind nicht Bestandteil der JRE. Bei den meisten untersuchten AR-Anwendungen ist es aber möglich, eine Erweiterung des Code-Parsers durchzuführen. Eine Analyse des Laufzeit-Verhaltens eines Programmes, das dies explizit verhindern will (z.B. Virenscanner, Verschlüsselungs-Software), sollte unmöglich oder zeitlich nicht vertretbar sein. Die vormals technische Hürde der dynamischen Analyse, kristallisiert sich in diesem Fall zu einer effektiven Limitation heraus.

Es gibt mehrere (nicht-exklusive) Richtungen, in die sich das semi-automatisierte AR weiterentwickelt. Eine Industrialisierung der akademisch entwickelten Methoden wird vermutlich die Wartungsbarriere minimieren. NDepends[11] und inCode[12] sind gute Beispiele dafür. Auch akademische Neuimplementierungen wie das MARPLE Project [FoZa11] setzen bereits erprobte AR-Methoden ein und versuchen diese zu verfeinern.

10 Der Release von J2SE 5.0 war am 30. September 2004

11 NDepends Website: http://www.ndepend.com/

12 InCode Website: http://www.intooitus.com/products/incode

7 Referenzen

Literaturverzeichnis

[ChCr90]: E. J. Chikofsky, J. H. Cross II; Reverse Engineering and Design Recovery: A Taxanomy; IEEE Software; Seiten 13-17; Ausgabe 1; Band 7; 1990

[Cifu94]: C. Cifuentes; Reverse Compilation Techniques; Queensland University of Technology; 1994

[MeEgGr03]: N. Medvidovic, A. Egyed, P. Grünbacher; Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery; ICSE 2003 - Proceedings of 2nd International Software Requirements to Architectures Workshop; Seiten 61-68; 2003

[IEEE00]: Architecture Working Group; IEEE Recommended Practice for Architectural Description of Software-Intensive Systems ; IEEE Computer Society; 2000

[WCCWWM95]: B.A. Wichmann, A.A. Canning, D.L. Clutterbuck, L.A. Winsborrow, N.J. Ward, D.W.R. Marsh; Industrial perspective on static analysis; Software Engineering Journal; Seiten 69-75; Ausgabe 2; Band 10; 1995

[NNH04]: F. Nielson, H. R. Nielson, C. Hankin; Principles of Program Analysis; Springer; 2004

[PiAlHa09]: Heidar Pirzadeh, Luay Alawneh, Abdelwahab Hamou-Lhadj; Quality of the Source Code for Design and Architecture Recovery Techniques: Utilities are the Problem ; 9th International Conference on Quality Software; Seiten 465-469; 2009

[GHJV94]: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Design

Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley; 1994
[Magg09]: Stefano Maggioni ; Design Pattern Detection and Software Architecture Reconstruction: an Integrated Approach based on Software Micro-structures ; Università degli Studi di Milano-Bicocca; 2009
[LaRa06]: Michele Lanza, Radu Marinescu; Object-Oriented Metrics in Practice; Springer Verlag; 2006
[RoMu02]: M. P . Robillard, G. C. Murphy; Concern Graphs: Finding and Describing ConcernsUsing Structural Program Dependencies; Proceedings of the 24rd International Conference on Software Engineering; Seiten 406-416; 2002
[MuNoSu95]: G. C. Murphy, D. Notkin, K. Sullivan; Software Reflexion Models: Bridging the Gap between Source and High-Level Models; SIGSOFT FSE; Seiten 18-28; 1995
[CHLW06]: S. Cook, R. Harrison, M. M. Lehman, P. Wernick; Evolution in software systems: foundations of the SPE classification scheme; Journal of Software Maintenance and Evolution: Research and Practice; Seiten 1–35; Ausgabe 1; Band 18; 2006
[LRWPT97]: M. M. Lehman, J. F. Ramil, , P. D. Wernick, D. E. Perry, W. M. Turski; Metrics and laws of software evolution-the nineties view; 4th International Software Metrics Symposium; Seiten 20-32; 1997
[HoRe92]: Susan Horwitz, Thomas Reps; The Use of Program Dependence Graphs in Software Engineering; International Conference on Software Engineering; Seiten 392-411; 1992
[Shne91]: B. Shneiderman; Tree visualization with Tree-maps: A 2-d space-filling approach; ACM Transactions on Graphics; Seiten 92-99; Ausgabe 1; Band 11; 1991
[LaDu03]: M. Lanza, S. Ducasse; Polymetric Views - A Lightweight Visual Approach to Reverse Engineering; IEEE Transactions on Software Engineering; Seiten 782-795; Ausgabe 9; Band 29; 2003
[DoPa06]: B. Dobing, J. Parsons; How UML is used; Communications of the ACM - Two decades of the language-action perspective; Seiten 109-113; Ausgabe 5; Band 49; 2006
[JRHZQ07]: M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins; UML 2 glasklar; Carl Hanser Verlag GmbH & CO. KG; 2007

[DeDuNi08]: S. Demeyer; S. Ducasse; O. Nierstrasz; Object-Oriented Reengineering Patterns; Square Bracket Associates; 2008
[McCo04]: Steve McConnell; Code Complete - A Practical Handbook of Software Construction; Microsoft Press; 2004
[HuTh99]: A. Hunt, D. Thomas; The Pragmatic Programmer; Addison-Wesley Professional; 1999
[Me97]: B. Meyer; Object-Oriented Software Construction; Prentice Hall; 1997
[Atos06]: Atos Origin; Method for Qualication and Selectionof Open Source software (QSOS); Atos Origin; 2006
[Crock06]: D. Crockford; ; The Internet Engineering Task Force; 2006
[Gir11]: T. Gîrba; The Moose Book; The Moose Association; 2011
[Chisel11]: The CHISEL Group; Shrimp User Manual; http://www.thechiselgroup.org/shrimp_manual; Inhalt: Anleitung für Creole; Letzter Zugriff: 02.08.2011 um 11:16
[BRZKCKCAL10]: Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr.; Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments; Proceedings of the 32nd International Conference on Software Engineering; Seiten 455-464; 2010
[JRM2011]: G. Murphy; SourceForge.net: Reflexion Model Eclipse Plugin; http://sourceforge.net/projects/jrmtool/; Inhalt: Website des jRMTools; Letzter Zugriff: 19.08.2011 um 23:00
[GitHub11]: GitHub Inc.; Top Languages - GitHub; https://github.com/languages; Inhalt: Statistik der verwendeten Programmiersprachen; Besucht am 19.09.2011 um 19:00
[LaIn11]: Arbinger Systems; Languae Usage Indicator; http://lui.arbingersys.com/index.html; Inhalt: Statistik der verwendeteten Programmiersprachen; Besucht am 19.09.2011 um 19:00
[BHTKK10]: S. Becker, M. Hauck, M. Trifu, K. Krogmann, J. Kofron; Reverse Engineering Component Models for Quality Predictions; 14th European Conference on Software Maintenance and Reengineering; Seiten 194-197; 2010
[Schr35]: E. Schrödinger; Die gegenwärtige Situation in der Quantenmechanik; Die

Naturwissenschaften; Seiten 807-812; Ausgabe 48; Band 23; 1935
[FoZa11]: F. A. Fontana, M. Zanoni; A tool for design pattern detection and software architecture reconstruction; Information Sciences; Seiten 1306-1324; Ausgabe 7; Band 181; 2011

Abbildungsverzeichnis

Tabellenverzeichnis

8 APPENDIX

8.1 QSOS-Template

Diese Tabelle beinhaltet die bei der Evaluation der AR-Tools verwendeten Criterions inklusive ihrer Beschreibungen. Reihen, die keine Einträge in den „Score X“ Zellen haben, sind *Sections* oder rein informativen Charakters.

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Generic section	Generic criteria from QSOS version 1.6			
Intrinsic durability	Intrinsic durability			
Maturity	Maturity			
Age		Less than 3 months	If between 3 months and 3 years	After 3 years

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Stability		Unstable software with numerous releases or patches generating side effects	Stabilized production release existing but old. Difficulties to stabilize development releases	Stabilized software. Releases provide bug fixes corrections but mainly new functionalities
History, known problems		Software knows several problems which can be prohibitive	No know major problem or crisis	History of good management of crisis situations
Fork probability, source of Forking		Software is very likely to be forked in the future	Software comes from a fork but has very few chances of being forked in the future	Software has very little chance of being forked. It does not come from a fork either
Adoption	Adoption by community and industry			
Popularity (related to: general public, niche, ...)		Very few users identified	Detectable use on Internet	Numerous users, numerous references
References		None	Few refences, non critical usages	Often implemented for critical applications

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Contributing Community		No community or without real activity (forum, mailing list, ...)	Existing community with a notable activity	Strong community: big activity on forums, numerous contributors and advocates
books		No book about the software	Less than 5 books about the software are available	More than 5 books about software are available, in several languages
Development leadership	Organisation and leadership of developments			
Leading team		1 to 2 individuals involved, not clearly identified	Between 2 and 5 independent people	More than 5 people
Management style		Complete dictatorship	Enlightened despotism	Council of architects with identified leader (e.g: KDE)
Activity	Activity of the project and around the software			

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Developers, identification, turnover		Less than 3 developers, not clearly identified	Between 4 and 7 developers, or more unidentified developers with important turnover	More than 7 developers, very stable team
Activity on bugs		Slow reactivity in forum or on mailing list, or nothing regarding bug fixes in releases note	Detectable activity but without process clearly exposed, long reaction/resolution time	Strong reactivity based on roles and tasks assignment
Activity on functionalities		No or few new functionalities	Evolution of the product driven by the core team or by user's request without any clearly explained process	Tool(s) to manage feature requests, strong interaction with roadmap

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Activity on releases		Very weak activity on both production and development releases	Activity on production and development releases. Frequent minor releases (bug fixes)	Important activity with frequent minor releases (bugs fixes) and planned major releases relating to the roadmap forcast
Industrialized solution	Industrialization level of the project			
Independence of developments		Developments realized at 100% by employees of a single company	60% maximum	20% maximum
Services	Services offering			
Training		No offer of training identified	Offer exists but is restricted geographically and to one language or is provided by a single contractor	Rich offers provided by several contractors, in several languages and split into modules of gradual levels

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Support		No offer of support except via public forums and mailing lists	Offer exists but is provided by a single contractor without strong commitment quality of services	Multiple service providers with strong commitment (e.g: guaranteed resolution time)
Consulting		No offer of consulting service	Offer exists but is restricted geographically and to one language or is provided by a single contractor	Consulting services provided by different contractors in several languages
Documentation		No user documentation	Documentation exists but shifted in time, is restricted to one language or is poorly detailed	Documentation always up to date, translated and possibly adapted to different target readers (end user, sysadmin, manager, ...)
Quality Assurance	Quality assurance process			

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Quality Assurance		No QA process	Identifies QA process but not much formalized and with no tool	Automatic testing process included in code's life-cycle with publication of results
Tools		No bug or feature request management tool	Standard tools provided (for instance by a hosting forge) but poorly used	Very active use of tools for roles/tasks allocation and progress monitoring
Packaging	Packaging for various operating systems			
Source		Software can't be installed from source without lot of work	Installation from source is limited and depends on very strict conditions (OS, arch, lib, ...)	Installation from source is easy
Debian		The software is not packaged for Debian	A Debian package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution

Criterion	*Description*	*Score 0*	*Score 1*	*Score 2*
FreeBSD		The software is not packaged for FreeBSD	A port exists but it has important issues or it doesn't have official support	A official port exists in FreeBSD
HP-UX		The software is not packaged for HP-UX	A package exists but it has important issues or it doesn't have official support	A stable package is provided for HP-UX
MacOSX		The software is not packaged for MacOSX	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution
Mandriva		The software is not packaged for Mandriva	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution
NetBSD		The software is not packaged for NetBSD	A port exists but it has important issues or it doesn't have official support	A official port exists in NetBSD

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
OpenBSD		The software is not packaged for OpenBSD	A port exists but it has important issues or it doesn't have official support	A official port exists in OpenBSD
RedHat		The software is not packaged for RedHat/Fedora	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution
Solaris		The software is not packaged for Solaris	A package exists but it has important issues or it doesn't have official support (e.g: SunFreeware.com)	The software is supported by Sun for Solaris
SuSE		The software is not packaged for SuSE	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution

Criterion	*Description*	*Score 0*	*Score 1*	*Score 2*
Windows		The project can't be installed on Windows	A package exists but it is limited or has important issues or just cover some specific Windows release (e.g: Windows2000 and WindowsXP)	Windows is full supported and a package is provided
BSD	BSD			
FreeBSD		The software is not packaged for FreeBSD	A port exists but it has important issues or it doesn't have official support	A official port exists in FreeBSD
Mac OS X		The software is not packaged for MacOSX	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution
NetBSD		The software is not packaged for NetBSD	A port exists but it has important issues or it doesn't have official support	A official port exists in NetBSD

Criterion	*Description*	*Score 0*	*Score 1*	*Score 2*
OpenBSD		The software is not packaged for OpenBSD	A port exists but it has important issues or it doesn't have official support	A official port exists in OpenBSD
Linux	Linux			
Debian		The software is not packaged for Debian	A Debian package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution
Mandriva		The software is not packaged for Mandriva	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution
Red Hat		The software is not packaged for RedHat/Fedora	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution
SuSE		The software is not packaged for SuSE	A package exists but it has important issues or it doesn't have official support	The software is packaged in the distribution

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Source		Software can't be installed from source without lot of work	Installation from source is limited and depends on very strict conditions (OS, arch, lib, ...)	Installation from source is easy
Unix	Unix			
AIX		The software is not packaged for AIX	A package exists but it has important issues or it doesn't have official support	A stable package is provided for AIX
HP-UX		The software is not packaged for HP-UX	A package exists but it has important issues or it doesn't have official support	A stable package is provided for HP-UX
Solaris		The software is not packaged for Solaris	A package exists but it has important issues or it doesn't have official support (e.g: SunFreeware.com)	The software is supported by Sun for Solaris

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Windows		The project can't be installed on Windows	A package exists but it is limited or has important issues or just cover some specific Windows release (e.g: Windows2000 and WindowsXP)	Windows is full supported and a package is provided
Exploitability	Exploitability level			
Ease of use, ergonomics		Difficult to use, requires an in depth knowledge of the software functionality	Austere and very technical ergonomics	GUI including help functions and elaborated ergonomics
Administration / Monitoring		No administrative or monitoring functionalities	Existing, functionalities but incomplete and or need improvement	Complete and easy-to-use administration and monitoring functionalities. Possible integration with external tools (e.g: SNMP, syslog, ...)
Technical adaptability	Technical adaptability			

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Modularity		Monolithic software	Presence of hight level modules allowing a first level of software adaptation	Modular conception, allowing easy adaptation of the software by selecting or creating modules
Code modification		Everything by hand	Recompilation possible but complex without any tools or documentation	Recompilation with tools (e.g: make, ANT, ...) and documentation provided
Code extension		Any modification requires code recompilation	Architecture designed for static extension but requires recompilation	Principle of plugin, architecture designed for dynamic extension without recompilation
Strategy	Project's strategy			
License	License			

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Permissiveness (only if user wants to become owner of code)		Very strict license, like GPL	Moderate permissive license located between both extremes (GPL and BSD) dual-licensing depending on the type of user (person, company, ...) or their activities	Very permissive like BSD or Apache licenses
Protection against proprietary forks		Very permissive like BSD or Apache licenses	Moderate permissive license located between both extremes (GPL and BSD), dual-licensing depending on the type of user (person, company, ...) or their activies	Very strict license, like GPL

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Copyright owners		Rights held by a few individuals or entities, making it easier to change the license	Rights held by numerous individuals owning the code in a homogeneous way, making relicense very difficult	Rights held by a legal entity in whom the community trusts (e.g: FSF or ASF)
Modification of source code		No practical way to propose code modification	Tools provided to access and modify code (like CVS or SVN) but not really used to develop the software	The code modification process is well defined, exposed and respected, based on roles assignment
Roadmap		No published roadmap	Existing roadmap without planning	Versionned roadmap, with planning and measure of delays
Sponsor		Software has no sponsor, the core team is not paid	Software has an unique sponsor who might determine its strategy	Software is sponsored by industry

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Strategic independence		No detectable strategy or strong dependency on one unique actor(person, company,sponsor)	Strategic vision shared with several other free and open source projects but without strong commitment from copyrights owners	Strong independence of the code team, legal entity holding rights, strong involvement in the standardization process
User interface				
Internationalisation support	Is the whole user interface internationalizable? Is Unicode supported? Are Right-to-Left languages supported (as for Hebrew and Arabic)?	The feature doesn't exist	Limited support	Fully supported
Skin-ability	Does the UI have a complete skin system? (a User/UserGroup customizable interface for instance)	The feature doesn't exist	Limited support	Fully supported
Prerequisite				
Storage support				

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
File system storage	The data persisted by the software can-be/is stored in (flat) files in the file system. Reminder: here, a higher score does not mean a better product nor the opposite, it really depends on your needs and non functional requirements.	The data persisted by the software can't be stored in flatfiles	The data persisted by the software can be stored in flatfiles, though it's not the default option	The data persisted by the software is only stored in flatfiles
Database				

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Relational database support	The data persisted by the software can-be/is stored in relational databases. Reminder: here, a higher score does not mean a better product nor the opposite, it really depends on your needs and non functional requirements. If the score is 1 or 2, do detail (in the comment section) the supported db (MySQL, PostgreSQL, Oracle, SQLite, other) and their version.	The software does not support any relational databases	The software supports a few relational databases	The software supports all (or most) of the market relational databases

Criterion	*Description*	*Score 0*	*Score 1*	*Score 2*
XML database support	The data persisted by the software can-be/is stored in XML databases. Reminder: here, a higher score does not mean a better product nor the opposite, it really depends on your needs and non functional requirements. If the score is 1 or 2, do detail (in the comment section) the supported db (BerkeleyDB , other) and their Version.	The software does not support any XML databases	The software supports a few XML databases	The software supports all (or most) of the market XML databases

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Object database support	The data persisted by the software can-be/is stored in object (oriented) databases. Reminder: here, a higher score does not mean a better product nor the opposite, it really depends on your needs and non functional requirements. If the score is 1 or 2, do detail (in the comment section) the supported db (ObjectDB, Versant, Cache, other) and their version.	The software does not support any object databases	The software supports a few object databases	The software supports all (or most) of the market object databases

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Revision control system support	The data persisted by the software can-be/is stored in revision control system. Reminder: here, a higher score does not mean a better product nor the opposite, it really depends on your needs and non functional requirements. If the score is 1 or 2, do detail (in the comment section) the supported db (CVS, SVN, ClearCase, other) and their version.	The software does not support any RCS	The software supports just one or a few RCS	The software supports all (or most) of the market RCS
Data warehouse	Does the system support data warehousing?	This feature doesn't exist	The feature exists with limitation	
GUI				

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
IDE compatibility	Describes general compatibility with common IDEs like Eclipse	No collaboration with any IDE build in	IDE specific formats (like Eclipse projects) are supported	Full IDE integration as plugin/addin
Data				
Input data	Assesses where the data is coming from and how it is aquired	It is not possible to import binaries or source code.	Import of source code is possible. As an alternative it is possible to use common models like	Import of binaries and source code is possible.
Export data	Specifies how useful the data is for use in other applications.	Export of data is not possible. The save files are not easyly parsable.	Save files are easyly parsable e.g. XML files	Export of data possible into a common format like EMF.
Report generation	Specifies if reports may be generated in a human readable format.	It is not possible to generate reports or print the data.	It is possible to print the diagrams.	There is a full report support. Diagrams are presented in a printable or HTML representation with the corrosponding data.

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Architectural recovery methods				
Design Pattern Recognition	Specifies if and how DPR is supported.	Design Pattern Recognition is not supported	Design Pattern Recognition is built in but the list of comprehended patterns is not extendable.	Design Pattern Recognition is built in and additional design patterns may be added.
Metrics	Specifies if and how Metrics are supported.	Metrics are not supported.	The metrics of the system under observation are generated and presented. They are not interpreted in an AR sense though.	The metrics of the system under observation are generated and presented. They are interpreted in an AR sense.
Program flow analysis	Lists how the information flow within the application may be presented	It is not possible to get information on the information flow	The information flow may be statically analysed. The results are presented in an usable way e.g. Sequence diagrams	The information flow may be analyzed at runtime. The results are presented in an usable way e.g. Sequence diagrams.

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Software evolution	Checks if the application has support for software evolution analyzations.	The concept of software evolution is not taken into account.	There is a built in functionality to view older versions of the system but no aggregation is done.	The evolution of the analyzed software is presented and aggregated information is extracted.
Reflexion models	Specifies if and how Reflexion models are supported.	There is no support for comparing expected and existing architecture.	A simple Reflexion model may be generated but it is non-interactive.	The Reflexion model may be analyzed in detail. For example the the facts used for constructing the edges are listed.
Concern graphs	Specifies if and how Concern graphs are supported.	Concern graphs are not supported.	Concerns can be specified. No visualization is done.	Concerns can be specified and are additionally visualized.
Design Flaws detection	Checks if the application supports Design Flaws detection	Design Flaws are not supported.	Predefined Design Flaws are supported. It is not possible to easily specify new flaws.	Predefined Design Flaws are supported. It is possible to easily specify new flaws.
Visualization				

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Dependency diagram support	Checks if the application supports the visualization Dependency diagram.	No dependency diagrams are generated	Dependency diagrams are generated.	Dependency diagrams are generated and the architecture is browsable.
Tree maps support	Checks if the application supports the visualization Tree map.	Tree maps are not supported.	Tree maps can be generated. The generation depends on predefined metrics. It is not possible to select own metrics.	Tree maps can be generated. The underlying metrics can be selected.
Documentability	Describes if it is possible to add comments and documentation directly into the diagrams during the AR process.	It is not possible to add documentation directly to the diagrams.	It is possible to add documentation in the meta data of the diagram.	Documentation may be added directly into the diagram e.g. it is possible to add comments to specific diagram nodes.
UML support	Specifies if the software supports UML as presentation type.	No UML support	Common structural diagrams like class diagrams are used as presentation.	Behavioral diagrams like sequence diagrams are used additionally to structural diagrams.

Criterion	***Description***	***Score 0***	***Score 1***	***Score 2***
Polymetric views	Checks if the application supports the visualization Polymetric view.	No polymetric views are supported.	Polymetric views are supported. The underlying metrics may not be changed.	Polymetric views are supported. The metrics used to create the view may be changed.
Call graph visualization	This criterion holds the information if it is possible to explore a visualized call graph.	It is not possible to generate and explore a call graph.	A call graph can be generated but not traversed.	A call graph can be generated and explored.
JAVA feature Support				
Generics	Are Generics supported and how?	No Generics support.	Generics are supported but no inheritance is taken into account.	Full Generics support including inheritance.
Annotations	Are Annotations supported?	No Annotation support.	-	Annotations are supported.

Criterion	*Description*	*Score 0*	*Score 1*	*Score 2*
JavaDoc	Is JavaDoc Supported?	No JavaDoc support. The comments are not visible in the diagrams or by mouse over. No indication that there are JavaDocs	There is an indication that comments are present.	JavaDoc comments are accessible in the diagram whether by mouse over or a seperate field/diagram node.
Support for other JVM languages	Describes if the tool supports other JVM languaes or only Java.	Only Java is supported	It is possible to extend the code parser to use other languages.	Bytecode is used for analysis. As a result all JVM languages are supported.
Supported Java version	Specifies which version of Java is maximal supported.	<=1.4	<=1.5	<=1.6
Spring support	Holds the information if Spring is supported.	Dependencies introduced by Spring are not recognized.	It is detected that hidden dependencies may be present.	Dependencies are resoluted and added correctly to the dependency model.