Fachbereich 3
Mathematik - Informatik
Universität Bremen

Diplomarbeit

Visualisierung von Software-Klonerkennung

Felix Beckwermert

17. Oktober 2006

Betreut von
Rainer Koschke* und Martin Gogolla†

*Prof. Dr. Rainer Koschke ist Leiter der Arbeitsgruppe Softwaretechnik an der Universität Bremen
†Prof. Dr. Martin Gogolla ist Leiter der Arbeitsgruppe Datenbanksysteme an der Universität Bremen

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe angefertigt habe und dabei keine anderen als die angegebenen Hilfsmittel verwendet habe.

Bremen, 17.10.2006

Felix Beckwermert

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1.

Einleitung

1.1. Hintergrund der Diplomarbeit

"The utopian idea of software development is that an elegant solution is created which solves the task efficiently and which can be easily extended and adapted to new requirements."(Rieger, 2005, [4])

Untersuchungen zu Folge sind 5–15% eines industriellen Software-Systems redundant [3, 23, 4]. Solche Redundanzen wirken sich nachteilig auf die Entwicklung eines Software-Systems aus, denn sie erhöhen den Wartungsaufwand. Änderungen an duplizierten Codefragmenten müssen an allen vorkommenden Stellen im Quellcode vorgenommen werden. Gleichzeitig erhöhen sie unnötigerweise die Größe des Systems und reduzieren damit die Übersichtlichkeit für den Softwareentwickler. Nicht zuletzt aus diesen Gründen zählen Redundanzen in Sofware-Systemen – so genannte *(Software-)Klone* – zu den *bad smells* der Softwaretechnik [2].

Häufigster Grund für das Auftreten von Klonen ist *Copy&Paste*-Programmierung. Hierbei wird der betreffende Codeabschnitt einer bereits implementierten Funktionalität, die an einer anderen Stelle nochmals benötigt wird, einfach dupliziert. Diese Vorgehensweise bezeichnet man auch als *Code Cloning*.

Neben einigen Tools, die den Anwender dabei unterstützen, duplizierten Code (semi-)automatisch zu entfernen oder diesen gleich gänzlich zu vermeiden, existieren mittlerweile vor

allem eine Reihe von Techniken zur Klonerkennung und darauf aufbauende Erkennungs-Systeme. Diese beschränken sich jedoch fast immer bei der Präsentation der gewonnen Daten auf eine rein textuelle Darstellung der Ergebnisse. In den meisten Fällen stellt es sich aber als schwieriges Unterfangen heraus, in solchen Listen einen Überblick über die selektierten Daten zu gewinnen oder die Ausprägung eines bestimmten Merkmales zu erkennen. Gleichzeitig ist die Auswertung der Daten langwierig und es besteht auf Grund der schlechte Übersicht außerdem die Gefahr falscher Schlussfolgerungen [3, 4].

1.2. Aufgabenstellung

Ziel dieser Arbeit ist es daher, adäquate Visualisierungsformen zur Darstellung der Ergebnisse verschiedener Erkennungs-Algorithmen (vgl. Abschnitt 2.2 auf Seite 14) zu konzipieren.

Neben einigen grundsätzlichen Anforderungen (vgl. Abschnitt 3.4 auf Seite 29) ist bei der Wahl der Visualisierungen insbesondere zu beachten, dass diese an geeigneter Stelle in `GraVis` zu integrieren sind.

Bei GraVis handelt es sich um ein grafisches Analyse-Werkzeug aus der `Bauhaus Suite` (vgl. Abschnitt 5.1.1 auf Seite 47) zur Benutzerinteraktion.

Bauhaus ist ein Software-System, dass Wartungsingenieure durch geeignete Werkzeuge bei der Rekonstruktion von Architektursichten auf Altsystemen unterstützt und so das Programmverstehen auf Quelltext- und Architekturebene erleichtert, um dadurch die Qualität und Effizienz eines Wartungsprozesses zu steigern [37]. Das System ist aber nicht ausschließlich auf die Unterstützung der Arbeit eines Wartungsingeneurs zugeschnitten. Die vorhandenen Analysen und zugehörigen Werkzeuge können ebenso im *forward-engineering* angewendet werden und sind außerdem für die Bewertung der Qualität eines Systems z.B. im Qualitätsmanagement oder bei der Bewertung einer zugekauften Software von Interesse.

Die gewählte Visualisierung sollte daher die unterschiedlichen Bedürfnisse der verschiedenen Benutzer des Systems berücksichtigen.

Die Aufgaben der vorliegenden Diplomarbeit wurden in folgenden Schritte unterteilt:

- Einarbeiten in die Materie und Auswahl der den Visualisierugen zu Grunde liegenden Metriken
- Vergleich bereits existierender Konzepte die mit der Visualisierung und Analyse unter anderem von Softwareklonen befassen
- Auswählen oder Entwickeln geeigneter Visualisierungsformen zur visuellen Analyse der Ergebnisse der vorhandenen Klonenerkennungstools
- Implementierung der jeweiligen Visualisierungen und Integration in GraVis

1.3. Aufbau dieser Arbeit

Die vorliegende Arbeit gliedert sich in sechs Kapitel.

Nach dieser Einleitung folgt ein Kapitel mit einer Erläuterung des theoretischen Hintergrunds der Thematik. Darin werden die gebräuchlichen Begrifflichkeiten vorgestellt, sowie ein Überblick über die grundlegenden Ansätze zur Klonerkennung gegeben.

Im 3. Kapitel werden die im Folgenden verwendeten Metriken aufgezeigt, sowie die verschiedenen Visualisierungs-Arten und -Möglichkeiten erläutert.

Im 4. Kapitel erfolgt ein Vergleich vorhandener Erkennungs- und Visualisierungs-Systeme.

Das 5. Kapitel beginnt mit einer Übersicht über den Aufbau des Bauhaus-Projektes, insbesondere wird eine detaillierte Darstellung zu GraVis und dem RFG gegeben. Im zweiten Teil des fünften Kapitels werden die gewählten Visualisierungen vorgestellt und ihre Integration in GraVis aufgezeigt.

Im letzten Kapitel wird eine Zusammenfassung der erarbeiteten Ergebnisse, sowie ein Ausblick auf mögliche Erweiterungen gegeben.

Kapitel 2.

Grundlagen: Softwareklone

Dieses Kapitel gibt eine Einführung in die zugrunde liegende Materie dieser Arbeit, eine Voraussetzung für das Verständnis der folgenden Abschnitte und Kapitel. Nach einer Definition und genaueren Betrachtung der gebräuchlichen Begrifflichkeiten wird ein kurzer Überblick über die verschiedenen grundlegenden Erkennungs-Algorithmen gegeben. Dabei liegt das Augenmerk aber weniger in einer vollständigen Detail-Beschreibung der Algorithmen als viel mehr in einem Vergleich der verschiedenen Ansätze und deren Eigenschaften. Für genauere Informationen zu den jeweiligen Algorithmen wird auf andere Veröffentlichungen verwiesen. Ein ausführlicher Vergleich der Techniken mit einer Evaluation der produzierten Ergebnisse findet sich in [1].

2.1. Begriffs-Definitionen

Unter einem *Klon* versteht man in der Softwaretechnik die redundante Implementation einer bereits vorhandenen Funktionalität innerhalb eines Software-System. Diese Redundanz ist – mehr oder weniger – direkt aus dem, dem System zu Grunde liegenden, Quelltext ersichtlich (vgl. Abschnitt 2.1.2).

Häufigster Grund für das Auftreten solcher Klone ist das Wiederverwenden bereits vorhandener Funktionalitäten durch einfaches Kopieren und Einfügen des betreffenden Quelltextes. Soll z.B. eine bereits vorhandene Funktionalität an einer anderen Stelle in gleicher oder ähnlicher Weise wiederverwendet werden, so wird diese an die betreffende Position kopiert und mit kleinen Anpassungen, Erweiterungen oder Modifikationen versehen. Dieses Vorgehen wird häufig auch als *Code Cloning* oder *Copy&Paste*-Programmierung bezeichnet [1, 3, 4]. Der Grund dafür ist zumeist, dass der jeweilige Entwickler den Aufwand einer Modularisierung und eventueller unabsehbarer Folgen auf den Rest des Systems scheut.

Ein weiterer Grund für die Entstehung von Klonen liegt in den Eigenschaften einiger Programmiersprachen, bzw. Compiler. Weist eine Programmiersprache nicht die Möglichkeit der Wiederverwendung durch Vererbung oder generische Konstrukte auf oder bietet der zugehörige Compiler kein *Inlining*, bleibt häufig keine andere Möglichkeit, als die Duplikation der jeweiligen Funktionalitäten, um das gewünschte Ziel zu erreichen [3].

Eher selten kann es auch durch andere Fehler, wie z.B. Konflikte beim *Merging* zweier Versionen, durch ein unzureichendes Versionskontroll-System (VCS), zu duplizierten Codefragmenten kommen.

Der größte Nachteil des so genannten *Code Clonings* besteht dabei im Mehraufwand bei Wartungsarbeiten, also bei Anpassungen und Fehlerkorrekturen im Lebenszyklus eines Software-Systems. Bei Änderungen einer bestimmten Funktionalität eines Systems müssen alle weiteren ähnlichen Vorkommen identifiziert, auf die Notwendigkeit einer Überarbeitung geprüft und ggf. ebenfalls angepasst werden.

Zudem entstehen beim Kopieren und anschließenden Modifizieren häufig kleinere subtile Fehler, die schlimmstenfalls erst im späteren Verlauf zum Tragen kommen und dann häufig

nur sehr mühselig zu identifizieren sind.

Besonders bei übermäßigem Gebrauch von *Copy&Paste*-Programmierung leidet außerdem die Übersichtlichkeit mit zunehmender Größe eines Software-Systems.

Nicht zuletzt aus diesen Gründen führt *Code Cloning* (auf Platz 1 von 22) die „Stink Parade of Bad Smells“ der Softwaretechnik an [2].

Zur Beseitigung von Duplikaten gibt es, wie bereits oben erwähnt, abhängig von der eingesetzten Programmiersprache und dem verwendeten Compiler, unterschiedliche *Refactorings* als „Gegenmaßnahmen“.

Neben den puren Erkennungs-Werkzeugen gibt es aber auch zunehmend (halb-)automatische Refactoring-Tools, die den Anwender bei der Beseitigung der gefundenen Duplikate unterstützen oder dies sogar komplett automatisieren.

2.1.1. Codefragment

Wie bereits eingangs erwähnt, wird der einer redundanten Funktionalität zu Grunde liegende Quelltextabschnitt (im folgenden auch als Codefragmente bezeichnet) als *Klon* bezeichnet.

Ein Code-Fragment $Frag_1$ ist demnach genau dann Klon eines Fragmentes $Frag_2$, wenn $Frag_1$ und $Frag_2$ die gleiche oder stark ähnliche Funktionalität ausdrücken. Umgekehrt ist $Frag_2$ ein Klon von $Frag_1$. Man spricht auch von *Duplikaten*.

Ein solches Fragment bezeichnet im Allgemeinen einen beliebigen Ausschnitt einer Quellcodedatei, dessen Start- und End-Punkte dementsprechend keinerlei Beschränkungen unterliegen.[1]

2.1.2. Klontypen

Abhängig von der Art der *Ähnlichkeit* der jeweiligen Codefragmente werden Klone in vier Typen unterschieden:

[1] Um jedoch die Anzahl der Vergleiche und damit den Aufwand bei der Klonerkennung zu reduzieren, treffen die verschiedenen Verfahren unterschiedliche Einschränkungen für die zu vergleichenden *Einheiten* – also, die einem Klon zu Grunde liegenden Codefragmente (vgl. Abschnitt 2.2.1 auf Seite 15).

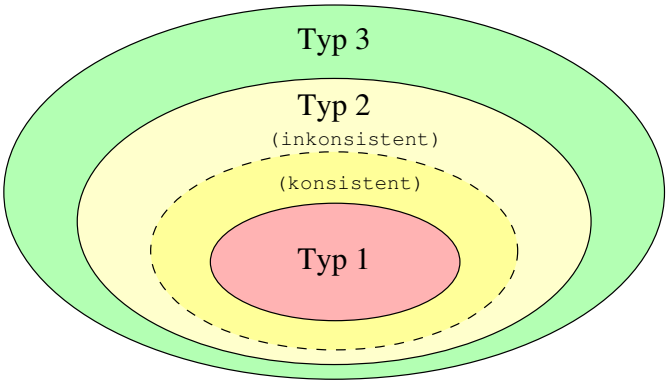


Abbildung 2.1.: Die Mengen der Typ 2-Klone umfasst die Menge der Typ 1-Klone und ist wiederrum in der Menge der Typ 3-Klone enthalten. Das Bild zeigt die im Folgenden verwendeten Farben für Klone vom Typ 1 bis 3.

Typ 1: exakte Kopie

Unter einem Typ 1-Klon versteht man die *exakte Kopie*, auch 1:1-Kopie oder *duplication*, eines Codefragmentes. Dabei wird in der Regel von vorhandenen *Whitespaces* und eventuell beinhalteten Kommentaren abstrahiert. Typ 1-Klone entstehen beispielsweise bei „manuellem“ *Inlining*.

Typ 2: Kopie mit parametrisierter Übereinstimmung

Bei einem Typ 2-Klon handelt es sich um eine Kopie mit *parametrisierter Übereinstimmung*. Neben *Whitespaces* und Kommentaren wird hierbei auch von Umbenennungen der Bezeichner abstrahiert. Dieser Klontyp entsteht in der Regel bei der Wiederverwendung einer Funktionalität, in dem diese kopiert wird mit anschließender Anpassungen der Bezeichner auf den jeweiligen Kontext, wie es beispielsweise bei generischen Funktion „von Hand“ der Fall ist.

Typ 2-Klone können zusätzlich in Klone mit konsistenter, bzw. inkonsistenter Umbenennung der Bezeichner unterteilt werden. Ist bei einem Typ 2-Klon in dem duplizierten Codefragment einer der Bezeichner umbenannt worden, so sind auch alle weiteren Vorkommen des Bezeichners auf die gleiche Weise umbenannt worden (vgl. Abb. 2.1.2). Das

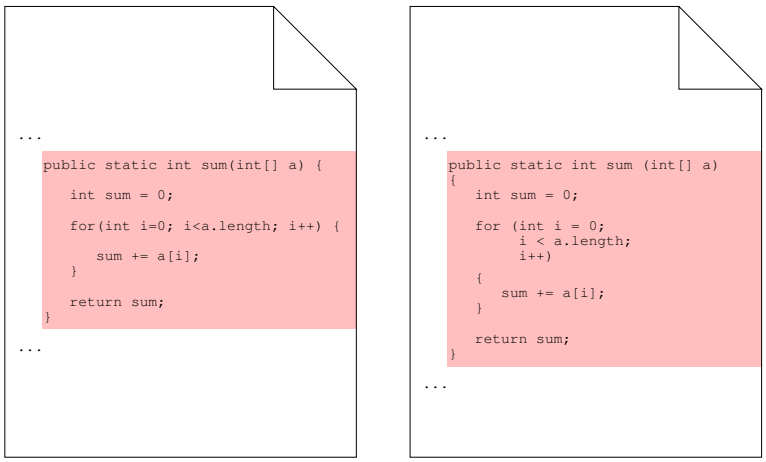


Abbildung 2.2.: Beispiel für einen Typ 1-Klon.

bedeutet insbesondere, dass die beiden betreffenden Codefragmente semantisch gleich sind. Bei einem Typ 2-Klon mit inkonsistenter Umbennung der Bezeichner ist dies dagegen nicht zwangsläufig der Fall[2].

Typ 3: Kopie mit weitergehenden Modifikationen

Ein Typ 3-Klon bezeichnet eine Kopie mit weitergehenden Modifikationen. Dieser Typ tritt häufig auf, wenn nur eine ähnliche Funktionalität benötigt wurde oder im Lebenszyklus der Software leichte Änderungen an einer Kopie vorgenommen wurden. Es handelt sich dabei meist um einen Typ 1- oder Typ 2-Klon, der z.B. durch Hinzufügen, Löschen oder Anpassen einzelner (Teile von) Anweisungen modifiziert wurde und somit „unterbrochen“ wurde. Man spricht auch von „zusammengesetzten“ Klonen, die sich dementsprechend häufig daran erkennen lassen, dass ein Typ 1- oder Typ 2-Klon durch einzelne ungleiche, bzw. unähnliche Anweisungsteile unterbrochen wird. Diese Eigenschaft nutzen auch die verschiedenen Verfahren zur Erkennung von Typ 3-Klonen aus.

[2]Dadurch kann ein sinnvolles Beseitigen des Klones – sofern man in einem solchen Fall überhaupt noch von einem Klon sprechen kann – durch Refaktorisierung unter Umständen unmöglich werden.

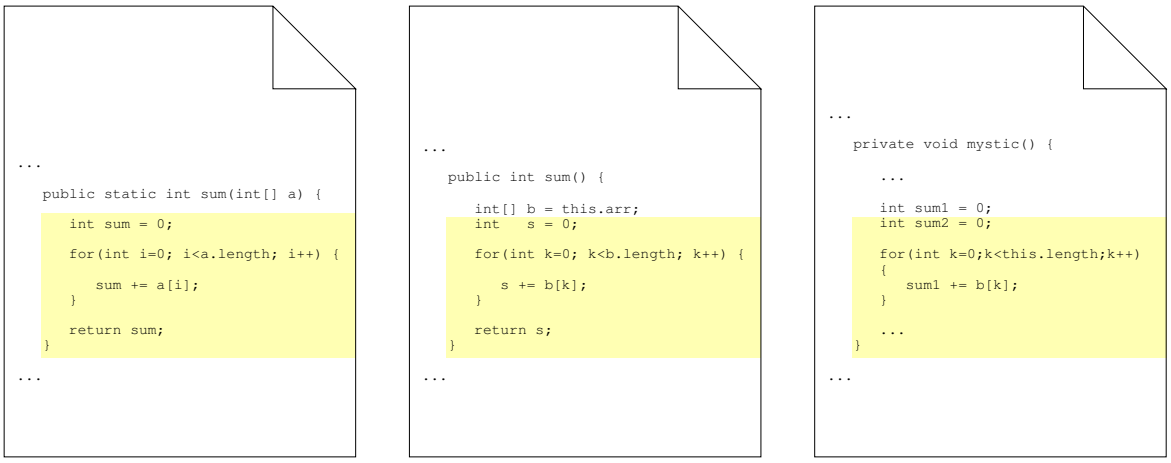
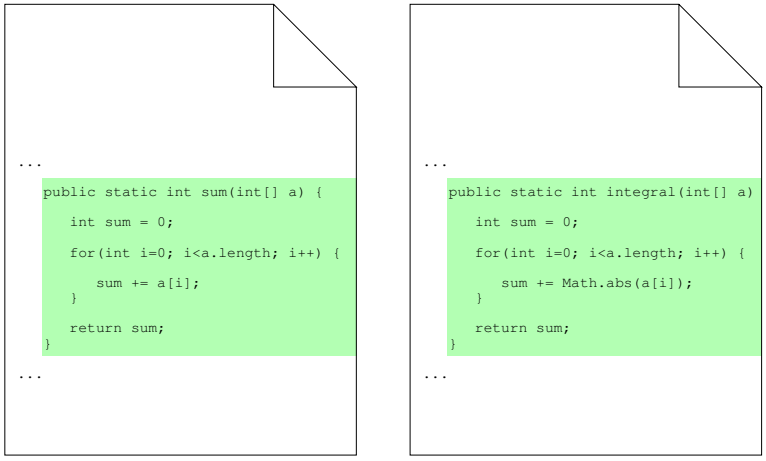


Abbildung 2.3.: Das Beispiel zeigt einen Typ 2-Klon mit inkonsistenter Umbennung der Bezeichner (links) und mit konsistenter Umbennung der Bezeichner (rechts) im Vergleich zum Fragment in der Mitte.

Typ 4: semantischer Klon

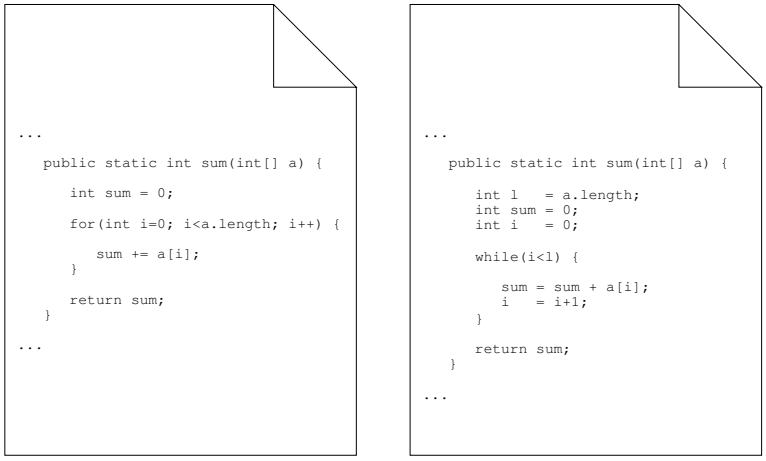
Der in der Literatur er selten erwähnte *semantische Klon* beschreibt zwei Codefragmente, die zwar die gleiche Funktionalität ausdrücken – da sie eben die gleiche Semantik beschreiben – aber syntaktisch nicht als Klon zu erkennen sind. Vielfach sind „Vorlagen“ für (primitive) Klone dieses Typs bereits in den Konstrukten unterschiedlicher Programmiersprachen enthalten. So können beispielsweise for-Schleifen grundsätzlich auch als *while*-Schleifen ausgedrückt oder die verkürzte Schreibweise des aus C, C++ oder Java bekannten *Inkrement*-Operators ++ häufig durch die ausgeschriebene Variante ersetzt werden.



```
...
   public static int sum(int[] a) {
      int sum = 0;
      for(int i=0; i<a.length; i++) {
         sum += a[i];
      }
      return sum;
   }
...
```

```
...
   public static int integral(int[] a)
      int sum = 0;
      for(int i=0; i<a.length; i++) {
         sum += Math.abs(a[i]);
      }
      return sum;
   }
...
```

Abbildung 2.4.: Beispiel für einen Typ 3-Klon.



```
...
   public static int sum(int[] a) {
      int sum = 0;
      for(int i=0; i<a.length; i++) {
         sum += a[i];
      }
      return sum;
   }
...
```

```
...
   public static int sum(int[] a) {
      int l   = a.length;
      int sum = 0;
      int i   = 0;
      while(i<l) {
         sum = sum + a[i];
         i   = i+1;
      }
      return sum;
   }
...
```

Abbildung 2.5.: Beispiel für einen Typ 4-Klon.

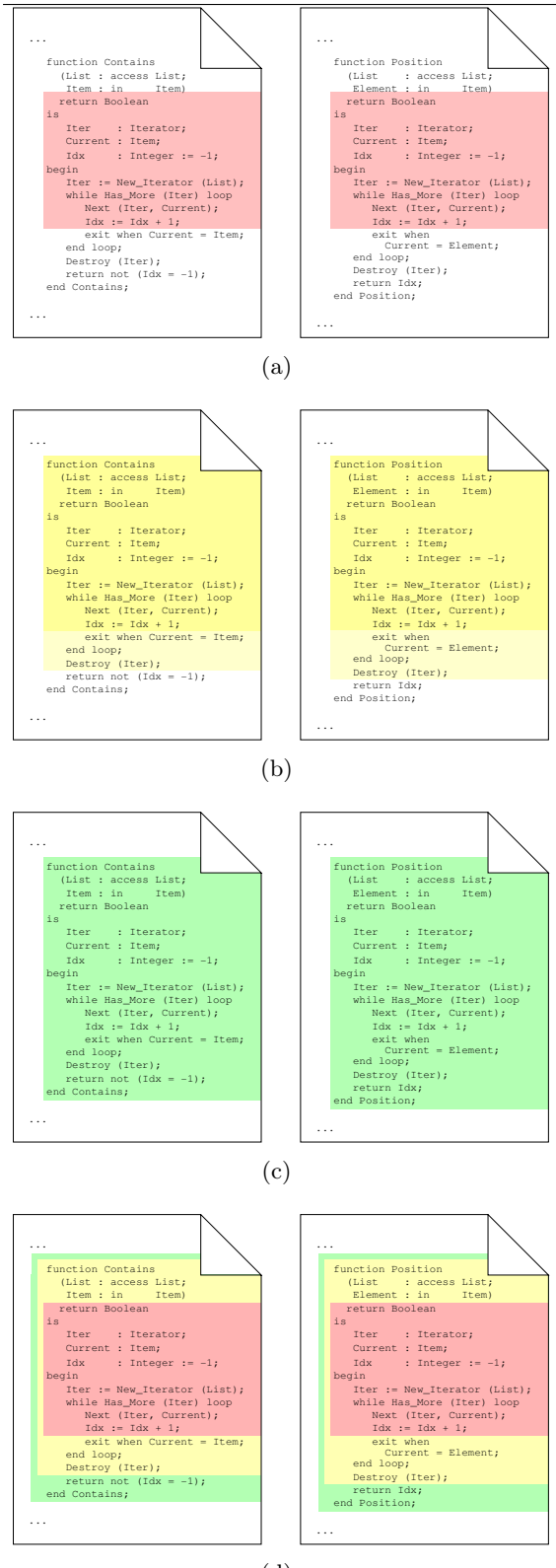


Abbildung 2.6.:
Das Beispiel zeigt die verschiedenen möglichen Klone, sowie deren Typ und die jeweils übereinstimmenden Codefragmente der Quelltextausschnitte zweier fast identischer Funktionen. Bei dem rot markierten Bereich (a) handelt es sich um eine exakte Kopie und damit um einen Typ 1-Klon. Bei dem gelb markierten Bereich (b) wurde dagegen ein Bezeichner konsistent umbenannt. Damit handelt es sich um einen Typ 2-Klon. Je nachdem, ob das eingesetzte Verfahren von *Whitespaces* abstrahiert, würde unter Umständen dabei der hell-gelb markierte Bereich mit erkannt werden oder eben nicht. Tatsächlich handelt es sich bei der ganzen Funktion jedoch um eine Kopie mit leichten Modifikationen vom Typ 3 (c), wie der grüne Bereich veranschaulicht. Da bei der Typ 3-Klon-Erkennung häufig einfach nahe beieinander liegende Typ 1- und Typ 2-Klone zu einem Typ 3-Klon zusammengesetzt werden, würden Klonerkennungsverfahren diesen aber nur äußerst selten erkennen. Die letzte Darstellung (d) zeigt nochmals einen Vergleich der unterschiedlichen erkannten Klone und ihrer Codefragemente.

2.1.3. Klonpaar

Ist ein Codefragment $Frag_1$ ein Klon eines Fragmentes $Frag_2$, so spricht man auch von dem Klonpaar $CP(Frag_1, Frag_2)$. Der Typ des Klonpaares ist dabei der Typ des Klones. Ein Klonpaar ist die kleinste eine Duplikation beschreibende Einheit.

Es handelt sich dabei um eine sowohl symmetrisch, als auch reflexive binäre Relation „Klonpaar“. Diese ist aber nur unter Umstände auch transitiv.

Existiert ein Klonpaar $CP_1(Frag_1, Frag_2)$ und ein Klonpaar $CP_2(Frag_2, Frag_3)$, so existiert nicht zwangsläufig auch ein Klonpaar $CP_3(Frag_1, Frag_3)$.

2.1.4. Klon-Klasse

Beschreiben mehrere Codefragmente eine gleiche, bzw. ähnliche Funktionalität, so werden diese zu einer Klonklasse zusammengefasst. Dabei handelt es sich um eine Äquivalenzklasse bezüglich der zwischen all diesen Klonfragmenten gemeinsamen Klonpaar-Relation. Das bedeutet insbesondere auf Grund der Transitiviät der Relation, dass jedes Fragment einer Klasse mit jedem Fragment ein Klon zu allen Fragmenten dieser Klasse ist (vgl. Abb. 2.1.4).

2.1.5. Klon-Cluster

Im Gegensatz zur Klonklasse werden in einem Klon-Cluster auch sich überschneidende Codefragmente zusammengefasst. Ist ein Fragement $Frag_1$ Klon eines Fragmentes $Frag_2$, das wiederum Teil eines Klones zweier Fragmente $Frag_3$ und $Frag_4$ ist (vgl. Abb. 2.1.4), so sind diese Fragmente alle Elemente des gleichen Klon-Clusters. Das bedeutet insbesondere auch, dass für ein weiteres $Frag_5$, welches Klon eines Teilfragmentes von $Frag_4$ und $Frag_3$ ist, nicht auch eine Klonbeziehung zu den Fragmenten $Frag_1$ und $Frag_2$ existieren muss. Es ist also möglich, dass ein Cluster Fragmente enthält, auf die keine Klonpaar-Relation definiert ist.

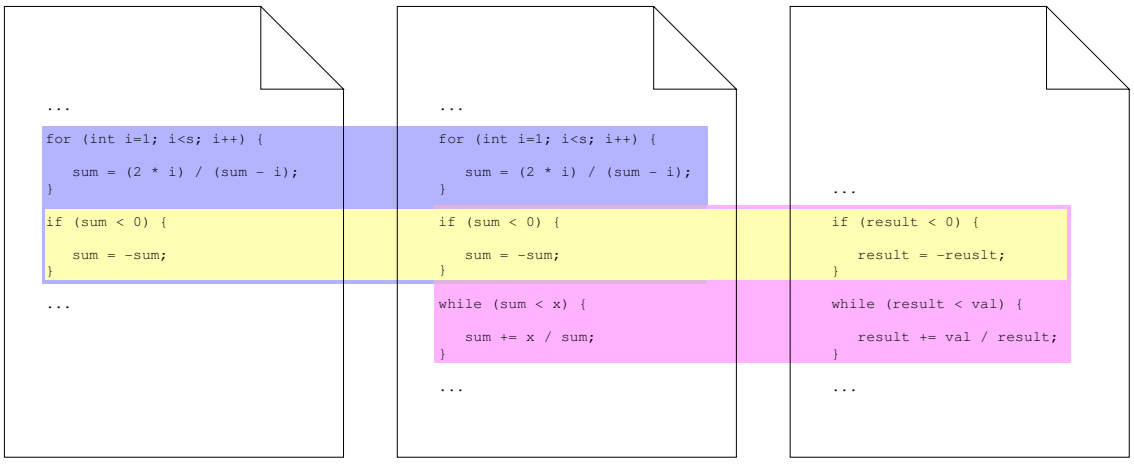


Abbildung 2.7.: Das Beispiel verdeutlicht den Unterschied zwischen einem Klon-Cluster und einer Klon-Klasse: Der blau gefärbte und der magenta gefärbte Klon bilden jeweils eine eigene (Äquivalenz-)Klasse. Aufgrund der Überschneidung der beiden Klone im gelb dargestellten Bereich sind diese aber trotzdem dem gleichen Klon-Cluster zugeordnet. Zusätzlich bildet der bei den Klonen gemeinsame gelb eingefärbte Abschnitt einen eigenen (Teil-)Klon mit eigener Klasse, der jeweils die Teilfragmente der ursprünglichen Klone angehören.

2.2. Erkennungs-Techniken

In den folgenden Abschnitten werden beispielhaft verschiedene Techniken zur Klonerkennung betrachtet. Neben einer kurzen Erläuterung des Verfahrens werden dabei vor allem einige Merkmale und daraus resultierende Einschränkungen, sowie weitere Vor- und Nachteile genauer betrachtet.

Insbesondere die folgenden Aspekte werden kurz erläutert:

- Granularität
- Abstraktionsebene
- Aufwand, bzw. Dauer
- Programmiersprachenunabhängigkeit
- erkannte Klontypen

- erkannte Klonart[3]

Die Granularität, sowie die Abstraktionsebene werden in den beiden folgenden Abschnitten näher betrachtet. Die *Dauer* eines Verfahrens wird maßgeblich durch die Anzahl und die Komplexität der Vergleiche bestimmt. Die Anzahl der potentiellen Vergleiche[4] ist dabei von der Granularität des jeweiligen Verfahrens abhängig, während die *Komplexität* sowie der hinzukommende Aufwand einer entsprechenden Vorverarbeitung des Quelltextes im Wesentlichen von der Abstraktionsebene abhängen. Ebenfalls stark abhängig von der Abstraktionsebene ist die Programmiersprachenunabhängigkeit. Die Klonart, die erkannten Klontypen, sowie die Klassifizierung[5] der erkannten Klone, ergibt sich dagegen einzig aus dem zu Grunde liegenden Algorithmus. Eine Erkennung von Typ 3-Klonen erfolgt dabei, sofern vorhanden, auf Grund dessen Eigenschaften (vgl. Abschnitt 2.1.2) meistens in einem abschließenden Schritt.

Am Ende des Abschnittes wird in der Tabelle 2.2.6 nochmals ein Überblick mit einem Vergleich über die Güte der verschiedenen Verfahren gegeben, der die Ergebnisse aus [1] und [3] wiedergibt.

2.2.1. Granularität

Unter der Granularität eines Verfahrens versteht man die „Größe“ der zu vergleichenden Einheiten. Sie bestimmt maßgeblich die Anzahl der potentiell nötigen Vergleiche. Bei feinerer Granularität wird der Programmcode in kleinere, aber dafür in eine größere Anzahl von Einheiten “portioniert“. Die Zahl der potentiellen Vergleiche und damit auch der Aufwand, bzw. die Laufzeit steigt quadratisch mit der Anzahl der Einheiten.

Sollen beispielsweise zwei zehnzeilige Codefragmente miteinander verglichen werden, so müsste ein zeilenorientiertes Verfahren – bei einem Vergleich jeder Zeile des einen Fragmentes mit jeder Zeile des anderen – potentiell 100 Vergleiche anstellen. Dagegen hätte ein

[3] Mit der *Klonart* wird unterschieden, ob gefundene Klone lediglich als Klonpaare geliefert werden oder die Ausgabe aufgeschlüsselt nach der Zugehörigkeit zu Klon-Klassen, bzw. Klon-Clustern erfolgt.

[4] Die Anzahl der tatsächlichen Vergleiche ist natürlich in erster Linie davon abhängig, welche Technik eingesetzt wird um einen quadratischen Aufwand aller Vergleiche zu vermeiden und auf das Nötigste zu reduzieren.

[5] Die *Klassifizierung* bedeutet in diesem Zusammenhang die Zuordnung des erkannten Klones zu seinem jeweiligen Typ. Die Klone eines Type bilden dessen *Typklasse*

tokenorientiertes Verfahren oder gar ein zeichenorientiertes Verfahren mit feinerer Granularität im Normalfall[6] wesentlich mehr Vergleiche zu erledigen.

Die hier betrachteten Verfahren setzen allerdings verschiedene Techniken zur Reduktion der Vergleiche und damit des quardratischen Aufwandes ein.

Abhängig von der Granularität eines Verfahrens ergeben sich zudem mögliche Beschränkungen auf die erkannten Klone, bzw. deren Codefragmente.

Üblicherweise werden die folgenden Granularitätsstufen unterschieden:

- Zeichen
- Zeilen
- Anweisungssequenzen
- Funktionen
- Module

2.2.2. Abstraktionseben

Die Abstraktionsebene dagegen beschreibt die „Art“ der zu vergleichenden Einheiten, bzw. das für einen Vergleich nötige „Wissen“ über diese.

Damit beispielsweise ein Verfahren Typ 2-Klone erkennen kann, muss es in geeigneter Weise von Bezeichnern abstrahieren. Voraussetzung dafür ist aber zunächst einmal, dass es diese überhaupt vom „Rest“ unterscheiden kann.

Der Aufwand zur Vorbereitung des untersuchten Quelltextes und damit die Laufzeit steigt dementsprechend mit zunehmender Abstraktionsebene an. Gleichzeitig sinkt die so genannte *Programmiersprachenunabhängigkeit*.

Folgende Abstraktionsebenen werden typischerweise unterschieden:

- Text
- Lexeme
- Syntax

[6] Voraussetzung ist natürlich, dass sich in einer Zeile mehrere Token, bzw. Zeichen befinden.

- Merkmale
- Programmabhängigkeiten

2.2.3. Verfahren nach Baker

Das zeilenorientierte Verfahren von Brenda S. Baker [6, 7], das in ihrem Tool `dup` zum Einsatz kommt, verwendet ein tokenbasiertes Pattern-Matching. Bei diesem Verfahren wird zunächst für jede Codezeile ein *Parameter-String* (abkürzend auch als *P-String* bezeichnet) generiert. Dieser besteht aus einem eindeutigen Nichtparametersymbol, dem *Funktor*[7], der die Struktur der Zeile wiedergibt, gefolgt von einigen *Parametersymbolen* bestehend aus den in der jeweiligen Zeile verwendeten Variablen.

Der so erzeugte konkatenierte P-String aller (Zeilen-)P-Strings, der den gesamten Programmcode repräsentiert, wird anschließend in einen *P-Suffix-Tree*[8] übertragen. Aus diesem lassen sich nun die Position, sowie Länge und Anzahl der Klone direkt bestimmen.

Durch Anwenden einer Funktion *prev* auf den P-String vor dem Aufbau des Suffix-Trees kann von Bezeichnern abstrahiert und somit Typ 2-Klone mit konsistenter Umbenennung erkannt werden. Die Funktion ersetzt dabei jeden Bezeichner durch die relative Position seiner letzten Verwendung.

Das Verfahren ist sehr schnell und liefert ansprechende Ergebnisse [1]. Es liefert Klonpaare vom Typ 1 und Typ 2 mit konsistenter Umbenennung und ist dabei weitgehend invariant gegen das Einfügen von *Whitespaces* [9]. Es existieren verschiedene Erweiterungen, mit denen sich sowohl Typ 3-Klone in einem separaten Schritt am Ende erkennen lassen [8] oder sich die Granularität des Verfahrens erhöhen lässt. Durch die niedrige Abstraktionsebene ist diese Erkennungstechnik weitgehend Programmiersprachenunabhängig [10].

[7] Der Funktor repräsentiert dabei *eindeutig* die *Struktur* der jeweiligen Codezeile. Zeilen mit gleicher Struktur werden dabei auf den gleichen Funktor abgebildet.

[8] Durch die Verwendung des P-Suffix-Trees wird dabei ein quadratischer Aufwand vermieden.

[9] Da die hier vorgestellte Version des Verfahrens zeilenorientiert arbeitet, ist sie dementsprechend nicht invariant gegen Einfügen von Zeilenumbrüchen. Es gibt allerdings verschiedene Erweiterungen, die dies ermöglichen.

[10] Prinzipiell müssen für die verschiedenen Programmiersprachen nur Bezeichner erkannt werden. Unter gewissen Umständen ist dazu aber bei einigen Programmiersprachen ein Parsing nötig.

Das Verfahren ist recht bekannt und gut verbreitet. Es wurde bereits einige Male erweitert oder mit anderen Verfahren kombiniert.

dup

Granularität	Zeilen
Abstraktionsebene	lexikalisch
erkannte Klone	Typ 1, Typ 2 Typ 3
Klassifikation	Typ 1–3
Klonart	Klonpaare

2.2.4. Verfahren nach Baxter et al.

Bei dem von Ira D. Baxter et al. [9] entwickelten und in ihrem Programm `CloneDR`™ implementierten Verfahren, werden Teilbäume des abstrakten Syntax-Baumes des jeweiligen Programms auf Gleichheit, bzw. Ähnlichkeit überprüft.

Dafür wird zunächst der AST des zu untersuchenden Quellcode erstellt und, um einen quadratischen Aufwand der Vergleiche aller Teilbäume zu vermeiden, alle Teilbäume ihrer Art (der Anweisung im jeweiligen Wurzel-Knoten) nach auf verschiedene Hash-Buckets verteilt.

Nun werden alle Teilbäume innerhalb eines Buckets mit Hilfe einer Ähnlichkeitsfunktion auf Gleichheit, bzw. Ähnlichkeit überprüft und ggf. der Menge der Klone hinzugefügt. Um im Ergebnis nur „maximal Klone“ zu erhalten, werden dabei eventuell bereits vorhandene Klone zwischen Unter-(Teil-)Bäumen wieder entfernt. Zur Erkennung von Klonen, die aus mehreren Anweisungen bestehen, ist es außerdem nötig in einem weiteren Schritt „Sequenzen von Klonen“ zu Einem zusammenzufassen und dann ggf. deren „Vater-Knoten“ erneut zu prüfen.

Abhängig von der gewählten Ähnlichkeitsfunktion erkennt das Verfahren Klonpaare der Typen 1, 2 und 3[11]. Eine Besonderheit des Verfahren ist, dass auch die Kommutativität von Operatoren berücksichtigt werden kann. Auf Grund des syntaxba-

[11] Die Typ 3-Erkennung erfolgt dabei in einem separaten Schritt am Ende.

sierten AST-Matchings ist das Verfahren relativ aufwändig. Da für die Erstellung des ASTs ein Parser für die jeweilige Programmiersprache notwendig ist, ist das Verfahren zudem wenig Programmiersprachenunabhängig. Allerdings wird auf diese Weise von jeglichen *Whitespaces*, sowie Kommentaren und Bezeichnern abstrahiert.

CloneDR™

Granularität	Anweisungen
Abstraktionsebene	syntaktisch
erkannte Klone	Typ 1, Typ 2 Typ 3
Klassifikation	Typ 1–3
Klonart	Klon-Kluster

2.2.5. Verfahren nach Mayrand et al.

Das Verfahren von Jean Mayrand et al. [13] erhebt Metriken um verschiedene Merkmale des zu Grunde liegenden Quellcodes zu vergleichen. Eine Vergleichsfunktion gibt dabei an, für welche Ausprägungen der gewählten Merkmale die betreffenden Codefragmente als gleich (Typ 1), bzw. ähnlich (Typ 2) zu werten sind. Eine Vermeidung des quadratischen Aufwands erfolgt dabei nicht.

Die Granularität, wie auch die erkannten Typen sind hierbei abhängig von der gewählten Vergleichsfunktion. Ebenso wird die Abstraktionsebene des Verfahrens durch die von der Funktion verwendeten Metriken bestimmt.

Da auf der Funktionsebene die meisten Metriken existieren, werden üblicherweise Klonpaare von Funktionen der Typen 1, 2 und 3 erkannt. Eine Klassifizierung erfolgt dabei nicht. Die Metriken unterteilen sich in die Bereiche „Name“, „Layout“, „Anweisungen“ und „Kontrollfluss“ und werden somit über den Quelltext selbst, den AST, sowie den Kontrollflussgraphen (KFG) erhoben.

Abhängig von den Metriken und der Vergleichsfunktion abstrahiert das Verfahren

im allgemeinen von *Whitespaces* und Kommentaren oder berücksichtigt diese explizit.

Granularität	Funktionen
Abstraktionsebene	PDG
erkannte	Typ 3, Typ 2
Klone	Typ 3
Klassifikation	Typ 3
Klonart	Klonpaare

2.2.6. Vergleich der Techniken

Die folgende Tabelle gibt einen Überblick über die oben erwähnten Eigenschaften der verschiedenen Verfahren und eine Bewertung ihrer Ergebnisse nach dem `Bellon Benchmark` [1]. Die einzelnen Angaben zur Güte der Verfahren stammen dabei aus [1] und [3].

	Baker	Baxter et al.	Mayrand et al.
Tool	dup	CloneDR™	
Abstraktionsebene	Token/Lexeme	AST	Metriken des PDG
Granularität	Zeilen[3]	Anweisungen	Funktionen[4]
erkannte Typen	Typ 1[3], Typ 2[6] (Typ 3[7])	Typ 1, Typ 2	Typ 1, Typ 2, Typ 3
Klassifikation	Typ 1, Typ 2 (Typ 3)	Typ 1, Typ 2	Typ 3
Klonart	Paare	Klassen	Paare
Aufwand/Dauer	++	-	–
Programmiersprachen-unabhängigkeit	+	-	–
Ergebnis			

Neben den hier vorgestellten Verfahren existiert eine Vielzahl weiterer. Die hier gewählten gehören jedoch zu den populärsten und decken zudem die verschiedenen Granularitäten und Abstraktionsebenen, sowie Techniken zur Vermeidung des quadratischen Aufwands ab.

Andere Veröffentlichungen verwenden oft ähnliche Ansätze, erweitern die hier aufgeführten Verfahren um spezielle Funktionalitäten oder kombinieren die verschiedenen Ansätze miteinander.

Kapitel 3.

Grundlagen: Softwarevisualisierungen

Im Folgenden wird nun der Begriff der Informationsvisualisierung und der Softwarevisualisierung kurz erläutert. Dabei wird auf die Anforderungen, die wichtigsten Eigenschaften und die Ziele einer Visualisierung hingewiesen. Dazu ist es wichtig, zunächst das Konzept von Metriken und insbesondere Softwaremetriken zu begreifen, da diese üblicherweise die einer Visualisierung zu Grunde liegenden Daten liefern.

3.1. Metriken

Das Maß für die Ausprägung eines bestimmten Merkmals von Objekten bezeichnet man auch als *Metrik*. Ein *Merkmal* bezeichnet dabei eine beliebige Größe oder Eigenschaft eines Objektes. Diese (empirische) Größe wird durch eine Metrik auf einen (Zahlen-)Wert einer Skala abgebildet [1].

Die Art der Skala ist abhängig von der Art der jeweils gemessenen Eigenschaft. Folgende Skalen werden unterschieden:

[1] Streng (mathematisch) gesehen ist eine Metrik eine Distanzfunktion zur Bestimmung der Distanz von Größen einer quantitativen Skala. In der Softwaretechnik wird der Begriff jedoch synonym für ein „beliebiges“ Maß verwendet und drückt damit auch die Abbildung qualitativer Merkmale aus.

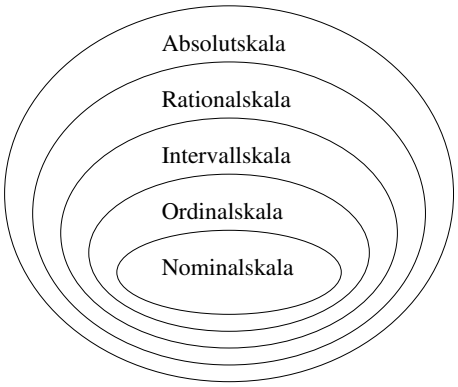


Abbildung 3.1.: Das Bild zeigt die verschiedenen Skalenniveaus. Höhere Skalen umfassen darunter liegende.

3.1.1. Nominalskala

Bei der Nominalskala handelt es sich um die schwächste Skala. Ein Merkmal ist *nominal*, bzw. *qualitativ* und damit dieser Skala zuzuordnen, wenn sich die möglichen Ausprägungen zwar voneinander unterscheiden, aber keine feste Ordnung auf diesen existiert. Die einzig mögliche Operation ist damit das Prüfen auf Identität. Die einzig mögliche Statistik ist die Häufigkeit.

Nominalskala	Kategorien (ohne Ordnung)
Operationen	Identität
Bedingungen	Symmetrie, Reflexivität, Transitivität
Statistik	Häufigkeit
Beispiele	Familienstand: ledig $\neq$ verheiratet
	Staatsangehörigkeit: deutsch $\neq$ amerikanisch

3.1.2. Ordinalskala

Von ordinalen Merkmalen wird gesprochen, wenn jede Merkmalsausprägung genau einer Kategorie zugeordnet wird und sich diese Kategorien zudem in eine feste Reihenfolge bringen lassen. Zulässige Operationen sind dementsprechend, neben dem

Vergleich auf Identität, der Vergleich auf Stärke, bzw. „Höhe“ der Ausprägung des Merkmals. Über die tatsächliche Differenz lässt sich dabei aber keine Aussage machen. Jede Ordinalskala ist auch eine Nominalskala.

Ordinalskala	Kategorien mit Ordnung
Operationen	(Größen-)Vergleiche
Bedingungen	Konnexivität, Transitivität
Statistik	Median
Beispiele	Größe: klein $\leq$ mittel $\leq$ groß

3.1.3. Intervallskala

Ein Merkmal ist der Intervallskala zuzuordnen, wenn sich die Merkmalsausprägungen quantitativ durch eine Zahl darstellen lassen und somit insbesondere ein bedeutungstragender Abstand zwischen zwei Ausprägungen bestimmt werden kann. Dieser Abstandswert lässt sich mit Hilfe einer Distanzfunktion exakt bestimmen. Zusätzlich zu den Operationen der Ordinalskala können so Differenzen und Summen gebildet werden. Eine Intervallskala ist immer auch eine Ordinalskala.

Merkmale dieser Skala werden auch als *kardinale*, *quantitative* oder auch *metrische* Merkmale bezeichnet.

Intervallskala	Zahlen
Operationen	Addition, Subtraktion
Statistik	Mittelwert, Standardabweichung
Beispiele	Jahreszahlen: $2007 - 1984 = 23$

3.1.4. Ratio(nal)skala

Für Merkmale, die sich auf einer Rationalskala, auch als Ratio- oder Verhältnisskala bezeichnet, abbilden lassen, existiert im Gegensatz zur Intervallskala ein absoluter

Nullpunkt. Neben den Operationen der Intervallskala ist zudem die Multiplikation und Division möglich. Dadurch lassen sich Aussagen über das Verhältnis von Merkmalsausprägungen, bzw. deren Werte, machen. Jede Distanzfunktion liefert auf Grund ihres definierten Nullpunktes somit Werte einer Rationalskala, wodurch unter anderem auch Aussagen über die Verhältnisse von Differenzen von Werten der Intervallskala möglich sind. Bei jeder Rationalskala handelt es sich auch um eine Intervallskala.

Rationalskala	Zahlen mit Nullpunkt
Operationen	Multiplikation, Division
Statistik	geometrisches Mittel, Korrelation
Beispiele	Länge: $100 = 2 * 50$

3.1.5. Absolutskala

Bei der Absolutskala ist die Maßeinheit der jeweils gemessenen Eigenschaft *natürlich gegeben*, das bedeutet, dass Merkmalsausprägungen nicht anders ausgedrückt werden können. Dieses ist beispielsweise immer dann der Fall, wenn es sich bei der gemessenen Eigenschaft um die Anzahl von Elementen einer Menge handelt. Die Anzahl ist natürlich bestimmt und kann daher nicht anders abgebildet werden.

Jede Absolutskala ist immer auch eine Rationalskala.

Absolutkala	Maßeinheit natürlich gegeben
Operationen	absoluter Vergleich
Beispiele	Anzahl der Frauen + Anzahl der Männer = Anzahl aller Personen

3.2. Softwaremetriken

Es gibt verschiedene Metriken für Software, die für unterschiedliche Personengruppen von Interesse sind. So unterstützen bestimmte Metriken den Entwickler bei der Wartung eines Systems oder geben dem Kunden oder dem Management Auskunft über die Qualität eines Software-Pakets.

3.3. Verwendete Metriken

Nachfolgend werden die im weiteren Verlauf dieser Arbeit verwendeten Software-Metriken vorgestellt, mit deren Hilfe sich Aussagen über die Art und Anzahl von dupliziertem Code innerhalb eines Systems treffen lassen.

Metriken auf syntaktischen Einheiten

Metriken auf syntaktischen Einheiten[2] werden im Kontext dieser Diplomarbeit – sofern nicht anders angegeben – wie nachfolgend beschrieben verwendet. Eine syntaktische Einheit bezeichnet dabei neben syntaktischen Elementen, wie Anweisungen und Funktionen, auch Module und Pakete oder Dateien und Verzeichnisse.

Lines of Code (LOC)

Die so genannten *Lines of Code* geben die Anzahl an Zeilen einer Einheit an. Die Metrik wird über den Sourcecode erhoben und es wird dabei weder von Kommentaren, noch von jeglichen *Whitespaces* abstrahiert. Auch eventuelle Spaltenangaben werden nicht berücksichtigt.

[2] Die vorgestellten Metriken beziehen sich dabei neben den syntaktischen Einheiten auch auf Einheiten der physikalischen und logischen Dekomposition. Im weiteren Verlauf ist in diesem Zusammenhang daher auch vereinfacht von „Einheiten“ die Rede.

Lines of Copied Code (LCC)

Die *Lines of Copied Code* geben die Menge an kopierten Zeilen innerhalb einer Einheit an. Abhängig vom Erkennungs-Verfahren, bzw. den gemeldeten Klonen, könne diese auch Kommentare oder leere Zeilen enthalten. Spaltenangaben werden auch hier nicht berücksichtigt.

Lines of Externally Copied Code (LEC)

Mit *Lines of Externally Copied Code* bezeichnet man die Anzahl an Code-Zeilen einer Einheit, die sie mit anderen Einheiten gemeinsam hat. Es handelt sich hierbei also um einen Teil der Anzahl der gesamt kopierten Codezeilen (LCC).

Lines of Internally Copied Code (LIC)

Unter den zuletzt vorgestellten *Lines of Internally Copied Code* versteht man die Anzahl der innerhalb einer Einheit kopierten Code-Zeilen.

Zu beachten ist dabei, dass auf Grund von Überschneidungen die Anzahl der extern (LEC) und intern (LIC) kopierten Codezeilen nicht zwangsläufig in der Summe mit der Anzahl der gesamt kopierten Code-Zeilen (LCC) übereinstimmen muss.

Metriken auf Klonen

Analog zu der Gesamtanzahl der kopierten Zeilen einer syntaktischen Einheit lässt sich diese Metrik auch auf Klone, sowie deren Aggregationen, die Klon-Klasse und Klon-Cluster, anwenden.

Lines of Copied Code (LCC)

Äquivalent zur Anzahl der kopierten Zeilen einer Einheit, gibt diese die Anzahl der *Lines of Copied Code (LCC)* für Klone und deren Aggregate die Zeilen der jeweils kopierten Codefragmente der beiden beteiligten Einheiten an.

Dabei ist zu beachten, dass die Anzahl der Zeilen der beiden Fragmente, je nach eingesetztem Verfahren und dessen Eigenschaften, beispielsweise bezüglich der Granularität, voneinander abweichen können.

3.4. Visualisierung von Informationen

Dieser Abschnitt befasst sich mit den Grundlagen der Informationsvisualisierung. Neben der Definition des Begriffs der Informationsvisualisierung und der Softwarevisualisierung werden die verschiedenen Ziele einer Visualisierung sowie die Unterschiede und Eigenschaften einfacher und komplexer Darstellungen erläutert.

3.4.1. Der Begriff der Informationsvisualisierung

Grundsätzlich wird jede rechnergestützte visuelle Repräsentation abstrakter Informationen, unabhängig von deren Quelle, als *Informationsvisualisierung* bezeichnet [16, 18].

Das generelle Ziel einer solchen Visualisierung ist dabei, eine gegebene Datenmenge in geeigneter Weise so darzustellen, dass diese besser „verstanden“ werden kann. Dieses Konzept beruht auf der menschlichen Neigung graphische Repräsentationen von räumlichen Anordnungen und Abhängigkeiten, sowie Detail-Informationen, wie Farbe, Form und Größe, genauer und schneller wahrzunehmen als in Form von abstrakten Begriffen [28].

Dagegen lassen sich prozedurale Informationen, logische Bedingungen und abstrakte verbale Konzepte besser in Form von abstrakten Begriffen beschreiben. Bei der

Gestaltung visueller Darstellungen ist zu beachten, dass zuerst die Struktur, Gestalt und Form einer Abbildung und dann einzelne Details wahrgenommen werden [28].

Im weiteren Verlauf dieser Arbeit bezeichnet die Informationsmenge, die Menge aller zu visualisierenden Eigenschaften von Objekten, während ein Beobachtungspunkt, ein Objekt, dessen Eigenschaften zu visualisieren sind, beschreibt.

3.4.2. Ziele einer Visualisierung

Zielklassen

Nach [15] können Informationen im Allgemeinen in drei Stufen unterschieden werden:

obere Stufe – Diese Stufe umfasst die gesamte Informationsmenge.

mittlere Stufe – Diese Stufe umfasst eine Teilmenge der Informationsmenge.

elementare Stufe – Diese Stufe beinhaltet einzelne Merkmalsausprägungen eines Beobachtungspunktes.

In Anlehnung an die beschriebenen Stufen lassen sich nach [24] drei Zielklassen einer Visualisierungs ableiten:

GLOBAL : Das Ziel ist einen Überblick über die Verteilung der Datenwerten in der gesamten Visualisierung zu erhalten.

LOKAL : Das Ziel ist das Erkennen von Zusammenhängen innerhalb eines Teilbereichs der dargestellten Informationen.

PUNKT : Das Ziel ist die Identifikation der Merkmalsausprägungen eines Beobachtungspunktes.

Adäquanz

Unabhängig von der gewählten Zielklasse ist nicht jede graphische Darstellung geeignet eine abstrakte Datenmenge, bzw. deren Eigenschaft und Zusammenhänge intuitiv zu repräsentieren. Eine Darstellung wird daher auch als *adäquat* bezeichnet, wenn sie die folgenden Kriterien erfüllt [25, 18]:

Expressivität Eine Visualisierung ist expressiv oder auch *expressiv adäquat*, wenn sie die in den Daten enthaltenen Informationen – und zwar nur diese – präsentiert.

Effektivität Eine Visualisierung ist effektiv oder auch *effektiv adäquat*, wenn es dem Betrachter möglich ist, die dargestellten Informationen intuitiv wahrzunehmen.

Angemessenheit Eine Visualisierung ist angemessen oder auch *leistungsadäquat*, wenn sie in angemessener Zeit erzeugt werden kann und zudem bei der Ausgabe die Ressourcen des jeweiligen Ausgabegerätes berücksichtigt.

Interaktionsdesign

Neben der Adäquanz ist es wichtig bei einer Visualisierung auf ein *effektives Interaktionsdesign* zu achten. Dabei sollte Shneidermans Leitlinie aus seinem *„visualization seeking mantra“* beachtet werden: *„Overview first, zoom and filter, then details on demand.“* [27].

Ausgehend von der Gesamtsicht der Informationsmenge soll der Benutzer mit Hilfe von geeigneten Interatktionsmechanismen die Darstellung irrelevanter Details (Projektion) und uninteressanter Beobachtungspunkte (Selektion) ausblenden können um so von der globalen über die lokale zur punktuellen Zielklasse alle Zielklassen abzudecken. Dazu ist es zudem nötig punktuell weitere Details einblenden zu können [27, 25, 18].

3.4.3. Einfache Abbildungen

In einfachen zweidimensionalen Darstellungen können Merkmale nach [15] und [21] auf verschiedene *visuelle Variablen* abgebildet werden (vgl. dazu auch [18]). Diese können demnach bezüglich ihrer Eignung zur Darstellung von Daten der verschiedenen Skalenniveaus wie folgt kategorisiert werden:

Datenstrukturierung	Visuelle Variable
Qualitative Daten (nominal)	Textur, Form, Sichtbarkeit, Vordergrund, Hintergrund, Richtung, Farbe
Qualitative Daten (ordinal)	Helligkeit, Sättigung, Größe, Fläche, Kontrast
Quantitative Daten (rational)	Position, Länge, Distanz

(Quelle: [21, 18]

Visuelle Attribute der ersten Gruppe können, da sie nur im Stande sind nominale Merkmalsausprägungen darzustellen (vgl. Abschnitt 3.1 auf Seite 23), nur zum Ausdrücken der Zugehörigkeit zu einer bestimmten Gruppe, bzw. Kategorie verwendet werden. Aufgrund der präattentiven Wahrnehmung des Menschen können bei moderater Verwendung bestimmte Gruppenzugehörigkeiten, z.B. Ausreißer, besonders hervorgehoben werden [28]. Dieses gilt insbesondere für der Verwendung von Farben.

Mit visuellen Attributen der zweiten Gruppe lassen sich dagegen ordinale Merkmalsausprägungen ausdrücken. Die Darstellungen der verschiedenen Kategorien muss deren definierten Ordnung entsprechen. Unter Umständen können auch Farben eine solche Ordnung der Elemente implizieren.

Quantitative Daten lassen sich dagegen lediglich auf Position, Länge und Distanz abbilden. Es sollte dabei davon abgesehen werden, diese visuellen Attribute für Daten einer nominalen Skala zu verwenden, da der Mensch dazu neigt, diese beim Betrachten stets in eine feste Rangfolge zu bringen [21].

Beispiele für einfache Abbildungen sind Abbildungen auf Farbe und Fläche oder Größe und Position, wie sie beispielsweise bei Diagrammen zum Einsatz kommen.

3.4.4. Komplexe Abbildungen

Gegenüber den einfachen können bei einer komplexen Abbildung mehrere Merkmale auf eine visuelle Variable abgebildet werden. Dadurch können immense Informationsmengen auf kompaktem Raum entstehen [18].

Durch ihren speziellen Verwendungszweck, haben komplexe Abbildungen aber häufig auch einige Nachteile:

- Oft kommt es vor, dass eine solche Darstellung nicht die gesamte Informationsmenge, die Merkmalsausprägungen aller Beobachtungspunkte, vollständig abbilden kann und damit die globale Zielklasse nicht erfüllt.
- Ebenso kann es vorkommen, dass sich nicht alle Ausprägungen eines Merkmals identifizieren lassen.
- In einigen Fällen können die Werte von Merkmalsausprägungen nicht bestimmt werden oder es können nicht alle Ausprägungen eines Beobachtungspunktes identifiziert werden.

Im Folgenden werden kurz einige, für den weiteren Verlauf relevante, der verschiedenen komplexen Visualisierungsformen erläutert.

Streckenzüge – Zu Streckenzügen zählen nach Illes beispielsweise Para-Histogramme. Dabei werden unterschiedliche Merkmale auf mehreren Achsen abgebildet, die entweder sternenförmig oder parallel angeordnet sind. Bei Para-Histogrammen wird eine solche parallele Anordnung um die Darstellung von Balken erweitert, die je nach Größe die Häufigkeit des Auftretens eines Merkmales angeben [18, 25].

Pixelbasierte Techniken – Bei pixelbasierten Techniken wird jedes Element der darzustellenden Menge über einen Pixel repräsentiert und die jeweilige Merkmalsausprägung auf dessen Farbwert abgebildet [18, 25].

Direkter Raumbezug – Bei einer Darstellung mit direktem Raumbezug werden Merkmale auf Detailattribute unmittelbar neben dem zugehörigen räumlichen Beobachtungspunkt abgebildet [18, 25].

Indirekter Raumbezug – Im Falle des indirekten Raumbezugs werden die Detailattribute dagegen in einer eigenen Visualisierung dargestellt. Dabei kann es sich wiederum um eine beliebige Darstellung handeln. Der indirekte Raumbezug wird dabei über eine visuelle Variable hergestellt [18, 25].

3.4.5. Softwarevisualisierung

Der Bereich der *Softwarevisualisierung* bezeichnet ein spezielles Teilgebiet der Informationsvisualisierung. Dieses Teilgebiet beschäftigt sich mit der grafischen Darstellung von Software-Systemen.

Bei den dargestellten Informationen handelt es sich um statische und dynamische Eigenschaften des Systems die typischerweise mit Hilfe von Softwaremetriken zur Visualisierung gebraucht werden [18].

Ziel einer Softwarevisualisierung ist in den meisten Fällen, den Benutzer beim „Verstehen“ oder Analysieren von Softwaresystemen, bzw. Algorithmen, zu unterstützen.

Neben dem häufigsten Einsatzgebiet zu visuellen Analysezwecken in der Softwareentwicklung, bzw. Softwarewartung, können Softwarevisualisierungen insbesondere auch zum Bewerten der Qualität eines Softwaresystems beispielsweise im Qualitätsmanagement herangezogen werden.

Kapitel 4.

Verwandte Arbeiten

Neben den bereits erwähnten, gibt es in der Literatur weitere Veröffentlichungen, die sich mit dem Thema der Visualisierung, unter anderem von Code Duplikationen beschäftigen.

Dazu zählen neben eng verwandten Arbeiten zur Klon-Visualisierung und Analyse von Software-System bzgl. Code-Duplikaten auch Ansätze zur Softwarevisualisierung im Allgemeinen.

Im Folgenden werden nun zunächst verschiedene dieser Visualisierungen vorgestellt und genauer betrachtet. Nach einer kurzen Erläuterung der allgemeinen Visualisierungs-Konzepte von *Dot Plots*, *Hasse-Diagramme* und *Polymetric Views*, werden einige der Visualisierungen aus [23] von Matthias Rieger, Stéphane Ducasse und Michele Lanza beschrieben, die in ihrer Arbeit fünf spezielle Verfahren zur Visualisierung „system-weiter Code Duplikation“ vorstellen. Diese basieren im Wesentlichen auf dem Konzept der erwähnten *polymetric views*.

4.1. Visualisierungen

4.1.1. Polymetric Views

Stéphane Ducasse und Michele Lanza stellten in [20] „die Kombination von Softwarevisualisierungen und Softwaremetriken durch Anreichern simpler Darstellung mit

metrischen Informationen“ vor, mit deren Hilfe – je nach gewählten Metriken und deren Abbildung – mehrere bestimmte Aspekte eines Systems gleichzeitig visualisiert werden können.

Bei der Darstellung handelt es sich um einen Graphen, auf dessen visuelle Variablen die verschiedenen Metriken abgebildet werden. Nach [20] können dazu die folgenden Eigenschaften der Knoten und Kanten verwendet werden:

- Breite der Knoten
- Höhe der Knoten
- Position der Knoten auf der X-Achse
- Position der Knoten auf der Y-Achse
- Farbe der Knoten
- Dicke der Kanten

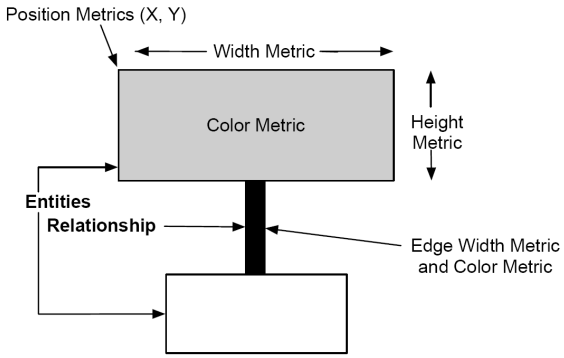


Abbildung 4.1.: Das Bild zeigt die visuellen Variablen einer Polymetric View. (Quelle: [23])

In einer weiterführenden Arbeit [23, 22] wurden unterschiedliche Verwendungen von *Polymetric Views* zur Visualisierung von Code-Duplikationen vorgestellt, die jetzt beschrieben werden.

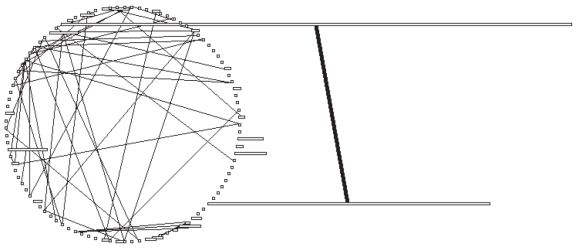


Abbildung 4.2.: Duplication Web
(Quelle: [23, 22])

4.1.2. Duplication Web

Beim *Duplication Web* [23, 22] handelt es sich um eine Polymetric View, die einen Überblick über die Klon-Situation des gesamten Systems geben soll. Dabei werden die Knoten, die die Dateien des Systems repräsentieren, kreisförmig angeordnet und über Kanten, die die Klone zwischen den jeweiligen Dateien abbilden, verbunden. Die Dicke der Kanten gibt dabei die Anzahl der kopierten Zeilen des jeweiligen Klons an, die Breite eines Knoten die Anzahl der internen kopierten Zeilen. Die Höhe eines Knotens ist dagegen fix.

Mit Hilfe des Duplication Webs soll der Entwickler einen Überblick über die Klon-Situation gewinnen. Aus ihr lässt sich schnell ermessen, wie stark der Klonbefall eines Gesamtsystems ist. Eventuelle Ausreißer mit besonders vielen intern oder extern kopierten Code-Zeilen fallen durch ihre Größe, bzw. Breite, respektive die Anzahl und Dicke ihrer Kanten auf.

Zu Problemen bei diser Polymetric View kann es kommen, wenn ein System sehr groß oder besonders stark mit Klonen belastet ist. Im ersten Fall wird entweder eine Zoom-Funktion notwendig oder die einzelnen Knoten und Kanten müssen entsprechend verkleinert werden. Auch eine Gruppierung nach Verzeichnissen ist möglich. Im zweiten Fall kommt es durch eine hohe Anzahl von Klonen zunehmend zu Überschneidungen und Verdeckungen, insbesondere bei den Kanten. In beiden Fällen leidet die Übersichtlichkeit sehr und es lassen sich kaum noch Detailinformationen bestimmen.

Duplication Web

Ansatz	polymetric view
Technik	Einfache Abb.
Abbildung	Knoten ↔ Dateien Kanten ↔ Klonen
Metriken	Knoten-Breite = LIC Kanten-Dicke = LCC
Datenmenge	Vollständig
Ziele	GLOBAL
Beispiel	Abb. 4.1.2
Literatur	[23, 22]

4.1.3. Clone Scatterplot

Bei einem *Clone Scatterplot* [23, 22] handelt es sich ebenfalls um eine Polymetric View. Dabei werden wiederum die Dateien des Systems durch die Knoten und Klone durch Kanten zwischen den jeweiligen Dateien eines Klones repräsentiert. Während die Kanten-Dicke wieder der Anzahl der kopierten Codezeilen eines Klons entspricht, besitzen die Knoten eine feste Größe. Die Position der Knoten wird dagegen von der Anzahl der Zeilen einer Datei (LOC) auf der X-Achse und der Anzahl der davon kopierten Zeilen (LCC) auf der Y-Achse bestimmt.

Aus der Darstellung lässt sich relativ leicht ein Überblick über die Qualität des Systems hinsichtlich der Anzahl der Klone im Verhältnis zu den Code-Zeilen gewinnen. Je näher ein Knoten der 45°-Linie kommt, um so mehr seiner Code-Zeilen sind kopiert. Ausreißer und besonders große Dateien mit vielen Klonen sind schnell zu erkennen.

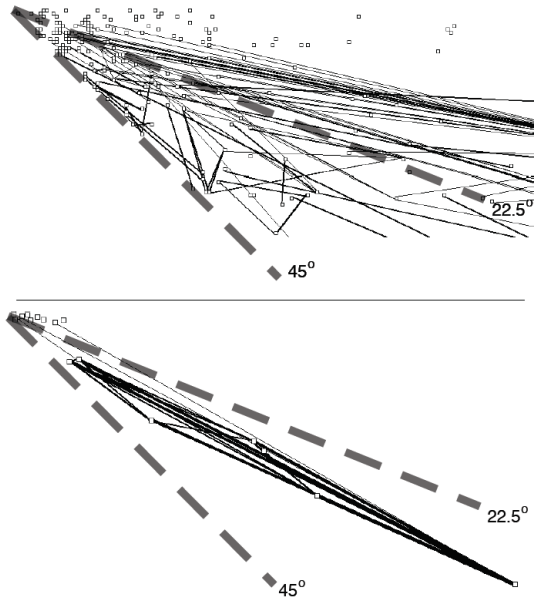


Abbildung 4.3.: Clone Scatterplot
(Quelle: [23, 22])

Clone Scatterplot

Ansatz	scatterplot, polymetric view
Technik	Einfache Abb.
Abbildung	Knoten ↔ Dateien Kanten ↔ Klonen
Metriken	Knoten-Pos. X = LOC Knoten-Pos. Y = LCC Kanten-Dicke = LCC
Datenmenge	Vollständig
Ziele	GLOBAL – LOKAL
Beispiel	Abb. 4.1.3
Literatur	[23, 22]

4.1.4. Duplication Aggregation Tree Map

Bei der *Duplication Aggregation Tree Map* [23, 22] handelt es sich um eine Mischform aus einer gewöhnlichen Tree Map und einer Polymetric View. Wie bei einer Tree Map wird dabei der Raum abhängig von einer Metrik zwischen den „Knoten“ aufgeteilt und hierarchisch aggregiert.

Im Unterschied zur gewöhnlichen TreeMap kommen jedoch zwei Metriken – eine für die X-Achse, die intern kopierten Codezeilen (LIC), und eine für die Y-Achse, die extern kopierten Codezeilen (LEC) – zum Einsatz. Zudem wird relativ viel Platz durch Freiraum „verschwendet“.

Die Breite eines Knotens ist in dieser Darstellung abhängig von den LIC, die Höhe von den LEC. Je höher der Anteil von LIC gegenüber den LEC, umso weiter wird der Knoten auf der X-Achse platziert. Je größer der Anteil der LEC gegenüber den LIC, umso weiter wird der Knoten entlang der Y-Achse verschoben.

Ein Knoten entspricht hier einer Datei. Sie werden dabei der hierarchischen Verzeichnisstruktur entsprechend aggregiert. Die Dateien eines Verzeichnisses werden

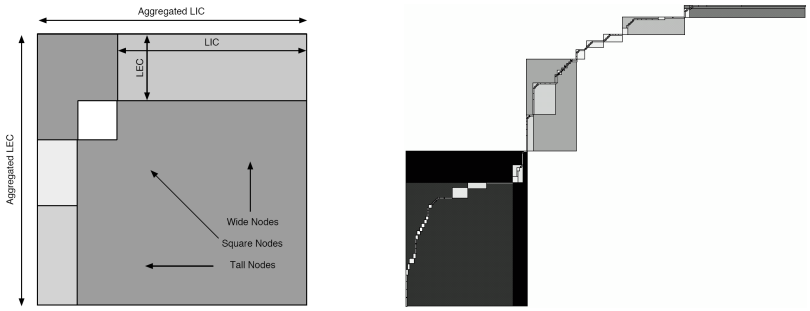


Abbildung 4.4.: Duplication Aggregation Tree Map
(Quelle: [23, 22])

zu einem „größeren“ Knoten, der das Verzeichnis abbildet, zusammengefasst. Dieses ist wiederum Teil des Knotens des überliegenden Verzeichnisses.

Anhand einer solchen Tree Map lassen sich schnell stark klonbelastete Teile eines Systems ausmachen. Die Darstellung ist zudem überlappungsfrei und bei größeren Systemen kommt es nicht zu einem Skalierungsproblem.

Duplication Aggregation Tree Map	
Ansatz	polymetric view, tree map
Abbildung	Knoten ↔ Dateien, Verzeichnisse
Technik	Einfache Abb.
Metriken	Knoten-Breite = LIC Knoten-Höhe = LEC Knoten-Farbe = LCC Knoten-Pos. X = LIC Knoten-Pos. Y = LEC
Datenmenge	Vollständig
Ziele	GLOBAL – LOKAL
Beispiel	Abb. 4.1.4
Literatur	[23, 22]

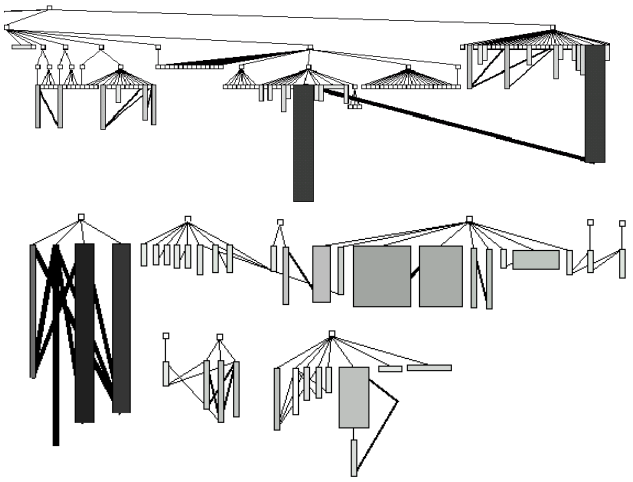


Abbildung 4.5.: System Model View
(Quelle: [23, 22])

4.1.5. System Model View

Bei der *System Model View* [23, 22] handelt es sich um eine Polymetric View, die die Verzeichnis-Struktur einer Applikation wiedergibt. Dabei werden Dateien als Knoten abgebildet, miteinander verbunden durch Kanten, die die Klone zwischen den jeweiligen Dateien repräsentieren. Die Knoten-Höhe wird dabei durch die Anzahl der extern kopierten Zeilen bestimmt, die Knoten-Breite durch die intern kopierten Zeilen. Die Breite der Kanten wird abermals durch die kopierten Zeilen (LCC) des jeweiligen Klons festgelegt.

Die einzelnen Dateien sind zudem über zusätzliche Kanten mit ihren Verzeichnissen verbunden und dementsprechend gruppiert. Die Verzeichnisse sind wiederum mit ihrem jeweils übergeordnetem Verzeichnis verbunden. Die Kanten und Knoten der Verzeichnisse besitzen dabei eine feste Größe.

Alternativ zur Darstellung nach der physikalischen Dekomposition, können die Knoten auch ihrer Vererbungshierarchie entsprechend angezeigt werden.

System Model View

Ansatz	tree layout, polymetric view
Abbildung	Knoten ↔ Dateien (und Verzeichnisse)
Technik	Einfache Abb.
Metriken	Knoten-Breite = LIC Knoten-Höhe = LEC Kanten-Breite = LCC
Datenmenge	Vollständig
Ziele	GLOBAL – LOKAL
Beispiel	Abb. 4.1.5
Literatur	[23, 22]

4.1.6. Tree Maps

Eine *Tree Map* [26] bildet Knoten eines Baumes auf rechteckige Flächen ab. Ein Rechteck der Tree Map entspricht einem Knoten des Baumes und ist Teil des Rechteckes des Vater-Knotens. Die Größe der Rechteckflächen wird über eine Metrik bestimmt. Die Rechteckfläche eines Knotens wird dabei unter den Flächen seiner Kind-Knoten aufgeteilt. Knoten mit stärkerer Ausprägung des jeweiligen Merkmals erhalten dabei im Verhältnis einen größeren Teil der Fläche des Vater-Rechtecks. Die Flächen der Knoten auf der untersten Ebene können in Abhängigkeit von einer weiteren Metrik eingefärbt werden.

In [17] wird eine Erweiterung um zusätzliche Metriken beschrieben. So können z.B. über die Textur, Lichteffekte oder per *Bump Mapping* weitere Metriken auf die Fläche der Blatt-Knoten abgebildet werden.

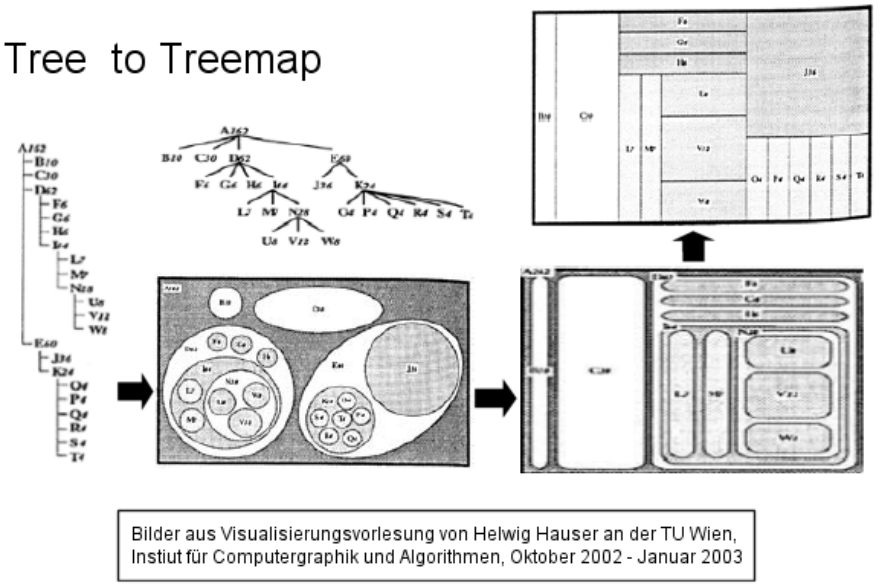


Abbildung 4.6.: Beispiele für verschiedene Tree Maps
(Quelle: [19])

Tree Map

Ansatz	Abb. von Knoten eines Graphen auf Rechtecke
Technik	Pixelbaisert
Abbildung	Knoten eines Baumes ↔ Rechteckflächen
Datenmenge	Vollständig
Ziele	GLOBAL, LOKAL, PUNKT
Beispiel	Abb. 4.1.6
Literatur	[26, 17]

Kapitel 5.

Konzeption

5.1. Umfeld

5.1.1. Bauhaus

`Bauhaus` [34] ist ein gemeinsames Projekt der Abteilung Programmiersprachen und Übersetzerbau des Instituts für Softwaretechnologie (`ISTE`) der Universität Stuttgart [29], der Arbeitsgruppe Softwaretechnik der Universität Bremen [30] und dem Fraunhofer Institut für Experimentelles Software-Reengineering (`FhG IESE`), Kaiserslautern [31]. Seit kurzem existiert auch eine kommerzielle Auskoppelung der Firma `Axivion GmbH` [33], die das System unter dem Produktnamen `Axivion Bauhaus Suite` – derzeit in der Version 5.3 – führt.

Das Projekt richtet sich vornehmlich an Wartungsprogrammierer mit dessen Hilfe diesen das so genannte Programmverstehen erleicht wird [37].

Wartungsprogrammierer stehen oft in der Pflicht, Anpassungen an *Legacy Systems* vornehmen zu müssen ohne das ihnen das jeweilige System näher bekannt wäre oder irgendeine ansprechende Art von Dokumentation existieren würde. Mit Hilfe der Bauhaus-Software können verschiedene Architektursichten rekonstruiert werden und verschiedene Analysen auf diese und dem zugrunde liegenden Code angewendet werden. Grundlage der Rekonstruktion ist dabei ausschließlich der Quellcode des Systems, das dieser meist die einzige vollständige und korrekte Beschreibung eines Systems bietet [37].

Derzeit umfasst das Projekt die folgenden Werkzeuge für verschiedene Dialekte und Versionen der Programmiersprachen C, C++, Java und Ada [32]:

- Architekturdarstellung
- Architekturprüfung
- Schnittstellenanalyse
- Metriken
- Klonerkennung
- Dominanzanalyse
- Zykluserkennung
- Komponentenerkennung
- Dead Code Detection

Im Folgenden wird ein kurzer Überblick über den Aufbau des Bauhaus Projektes gegeben. Dabei werden einzelne, für diese Arbeit relevante, Komponenten genauer beschrieben. Für detaillierter Informationen sei an dieser Stelle auf die einschlägige Literatur verwiesen [32, 34, 38, 37].

System-Aufbau

> *„Der Quelltext ist oft der einzige Informationsträger, der ein Altsystem wirklich verlässlich beschreibt.“* (Eisenbarth, 1999, [37])

Um unter anderem architekturbezogene Analysen in Bauhaus überhaupt erst zu ermöglichen, wird mit einem so genannten *language front end* zunächst aus dem jeweiligen Quellcode des zu analysierenden Systems ein Zwischencode – die *Intermediate Language (IML)* – erzeugt. Da es sich hierbei um eine weitgehend programmiersprachenunabhängige Darstellung handelt, wird die Integration weiterer Analysen erheblich vereinfacht und es besteht eine hohe Wiederverwendbarkeit bereits vorhandener Analysen. Zudem wird durch hohe Abstraktion bei gleichzeitiger größtmöglicher Quellnähe ein weitreichendes Spektrum möglicher Analysen abgedeckt .

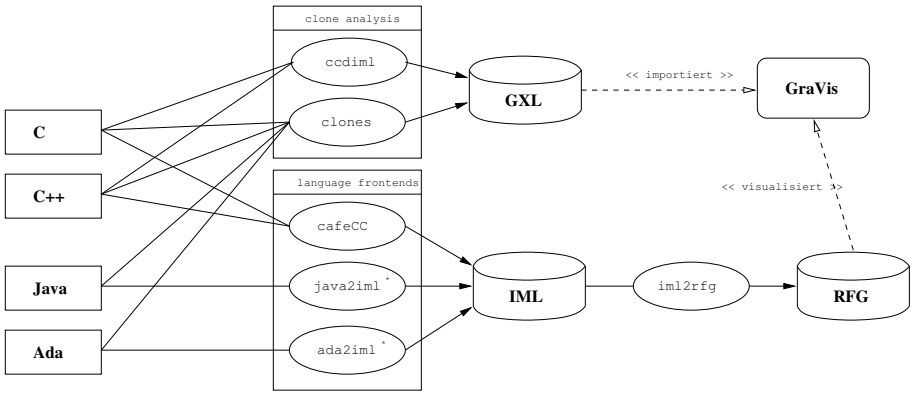


Abbildung 5.1.: Das Beispiel zeigt die wichtigsten Komponenten des Bauhaus Projektes.

Die IML stellt im Wesentlichen eine spezielle Form eines abstrakten Syntax-Baums, angereichert mit Kontrollfluss- und Datenfluss-Informationen dar.

Im nächsten Schritt wird aus dem so gewonnenen Zwischencode der *Resource Flow Graph (RFG)* abgeleitet. Dabei handelt es sich um eine stark abstrahierende Zwischendarstellung. Diese besteht aus einem hierarchischen Graphen, der architekturell relevante Elemente[1] als attributierte Knoten und Beziehungen zwischen diesen als ebenfalls attributierte Kanten abbildet. Die Knoten werden dabei der physikalischen und logischen Dekomposition des Systems entsprechend zu zusammengesetzten Komponenten aggregiert.

Über Teilgraphen werden zudem verschiedene Sichten (*Views*) auf den RFG ermöglicht[2]. Zur Visualisierung des RFGs bzw. der verschiedenen Views kommt das Graphenvisualisierungstool `GraVis` zum Einsatz.

Bei `GraVis` handelt es sich um ein grafisches *frontend* zur Anzeige und Bearbeitung des RFGs. Neben der Darstellung der verschiedenen Sichten ermöglicht dieser auch das Hinzufügen, Entfernen oder Modifizieren von Knoten, Kanten oder einzelner Attribute.

Außerdem können verschiedene auch über *plugins* eingebundene Analysen direkt

[1] z.B. Dateien, Typen, Routinen, ...
[2] z.B. Modul-Hierachie, Aufruf-Graph, ...

aus `GraVis` gestartet und deren Ergebnisse mit den bereits integrierten Werkzeugen dargestellt werden.

5.1.2. Der RFG

Der RFG bildet den Ausgangspunkt aller Visualisierungen in GraVis. Da er aus der IML abgeleitet wird, ist er dementsprechend programmiersprachenunabhängig. Die Knoten des RFG bilden dabei globale Programmentitäten sowie Objekte der physikalischen oder logischen Dekomposition eines Systems ab. Kanten beschreiben Beziehungen zwischen diesen Entitäten, Attribute auf Knoten und Kanten drücken zudem bestimmte Eigenschaften der abgebildeten Elemente aus.

GraVis verwendet den RFG als Datenmenge für die integrierten Visualisierungen. Zudem lassen sich Analysen auf dem RFG ausführen.

Um nicht immer auf der gesamten Knoten- und Kanten-Menge zu operieren, wurde das Konzept der *Views* integriert. Dabei handelt es sich um Teilgraphen, die nur die in einem gewissen Kontext oder für eine bestimmte Aufgabe relevante Teilmengen der Knoten und Kanten darstellen.

Neben einigen bereits vordefinierten Views münden die Ergebnisse der verschiedenen Anaylsen meist in einer neuen View. Zusätzlich können auch eigene Views erstellt, bzw. importiert oder vorhandene modifiziert werden. GraVis stellt dazu verschiedene Funktionalitäten z.B. zum Kopieren, Einfügen, Ein- und Ausblenden von Knoten, Kanten und Attributen einer View bereit[3]. Bei den vordefinierten Views, die bereits nach dem Laden eines RFGs zur Verfügung stehen, handelt es sich um die folgenden Sichten (vgl. Abb. 5.1.3):

- Base
- File
- Environment
- Call

[3]Die bereits vordefinierten Views können jedoch nicht modifiziert werden.

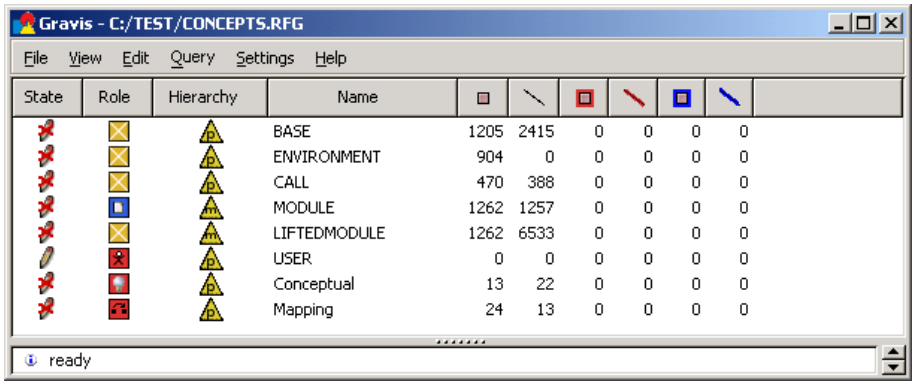


Abbildung 5.2.: Die Abbildung zeigt die Workbench mit den verschiedenen RFG Views. (Quelle: [?])

- Module
- Liftedmodule

5.1.3. GraVis

Nach dem Starten von `GraVis` öffnet sich die so genannte *workbench*. Wie der Name bereits impliziert, handelt es sich hierbei um das Hauptfenster, von dem alle weiteren Aktionen ausgehen. Es bietet die grundsätzlichen Funktionen der RFG-Verwaltung. Dazu gehören unter anderem das Laden und Speichern eines RFGs, das Ausführen von Analysen und das Öffnen von Dialogen mit weiteren Funktionalitäten. Die verschiedenen Funktionen sind über ein Menü zugänglich. Eine Statuszeile zeigt zusätzliche Informationen an, während in der Mitte die verschiedenen Sichten (*Views*) des aktuell geladenen RFGs dargestellt werden (vgl. Abb. 5.1.3).

Graph Window Über einen Doppelklick auf eine der Sichten in der Workbench öffnet sich ein *Graph Window*, in dem die Knoten und Kanten, sowie ggf. angezeigte Attribute des zu dieser Sicht gehörenden Teilgraphen des RFGs hierarchisch angeordnet angezeigt werden (vgl. Abb. 5.1.3).

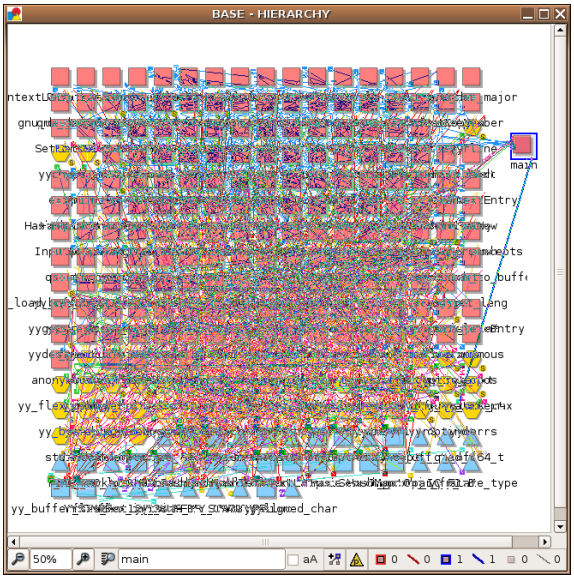


Abbildung 5.3.: Die Abbildung zeigt das Graph Window einer `Base View`.

Ein Knoten des *visuellen Graphen* entspricht dabei einem Knoten im RFG. Gleiches gilt für Kanten und Attribute. Die verschiedenen Knotentypen des RFGs werden dabei auf verschiedene Formen mit unterschiedlichen Farben der Knoten des visuellen Graphen abgebildet. Die dargestellten Kanten werden ebenfalls abhängig vom korrespondierenden Typ im RFG eingefärbt und die Richtung der Kante im RFG mit kleinen Rechtecken am jeweiligen Ende markiert (vgl. Abb. 5.1.3). Die Position der Knoten wird über *layouts* bestimmt, die vom Benutzer individuell ausgewählt werden können. Spezielle Layout-Algorithmen greifen dabei auf Informationen über die Beziehungen der Knoten untereinander – also deren Kanten – zurück.

In dem Graph Window lassen sich mit der linken Maustaste Knoten *selektieren*, bzw. *markieren* oder auch per *Drag&Drop* innerhalb des Fensters frei verschieben. Über das Kontextmenü der rechten Maustaste kann zudem beispielsweise das *Layout* des Graphen im aktuellen Fenster (*Graph Window*) geändert, genauere Informationen zu Knoten oder Kanten angezeigt oder weitere Tools wie z.B. die *Metric Box* geöffnet werden (vgl. Abb. 5.1.3). Selektierte, beziehungsweise markierte Knoten und Kanten

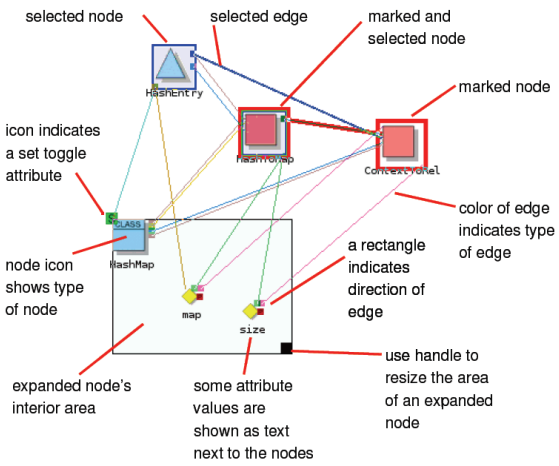


Abbildung 5.4.: Die Abbildung zeigt die verschiedenen visuellen Elemente eines Graph Window.
(Quelle: Bauhaus Guided Tour)

werden dabei in allen aktuell geöffneten Fenstern ausgewählt.

Mit einer integrierten Zoom- und Filter-Funktionalität, die das Ein- und Ausblenden von Informationen erlaubt, erfüllt das Graph-Window alle Ziele einer Visualisierung. Details lassen sich auf Wunsch über einen zusätzlichen Dialog im Kontextmenü anzeigen.

Metric Box Bei der Metric Box handelt es sich um ein Tool zur visuellen Analyse metrischer Informationen. Es können dabei beliebig viele Plots innerhalb einer Metric Box dargestellt werden. Ein Plot ist ein einfaches zweidimensionales Punktdiagramm, dessen Achsen verschiedene Metriken zugeordnet werden können (vgl. Abb. 5.1.3). So lassen sich Aussagen über das Verhältnis und den Zusammenhang verschiedener Eigenschaften eines Systems treffen. Die Bedienung der Metric Box erfüllt dabei alle Anforderungen an eine effektive Benutzerinteraktion.

Code Browser Neben der Metric Box kann über das Kontextmenü des Graph Windows der Quelltext zu dem unter dem Cursor befindlichen Knoten bzw. der Kante angezeigt werden. Es öffnet sich ein kleiner Editor, der die betreffende Quelldatei

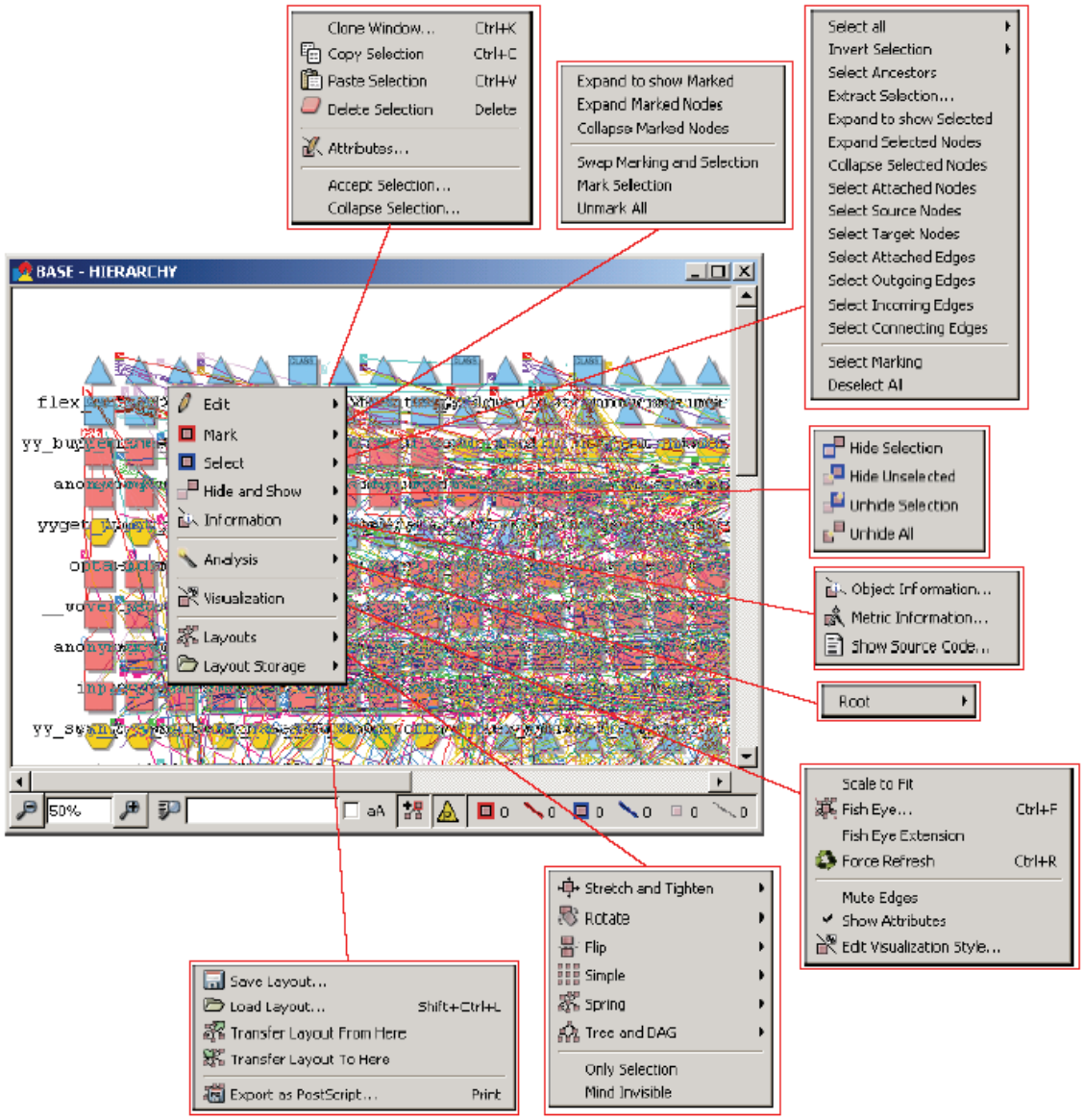


Abbildung 5.5.: Die Abbildung zeigt die verschiedenen Einträge im Kontextmenü eines Graph Windows.
(Quelle: Bauhaus Guided Tour)

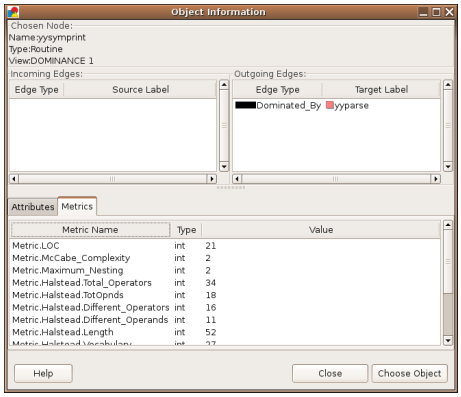


Abbildung 5.6.: Die Abbildung zeigt den Object Information Dialog, der neben allgemeinen Informationen zu Knoten, bzw. Kanten auch alle Metrik-Informationen und sonstigen Attribute, sowie verbundene Kanten, bzw. Knoten anzeigt.

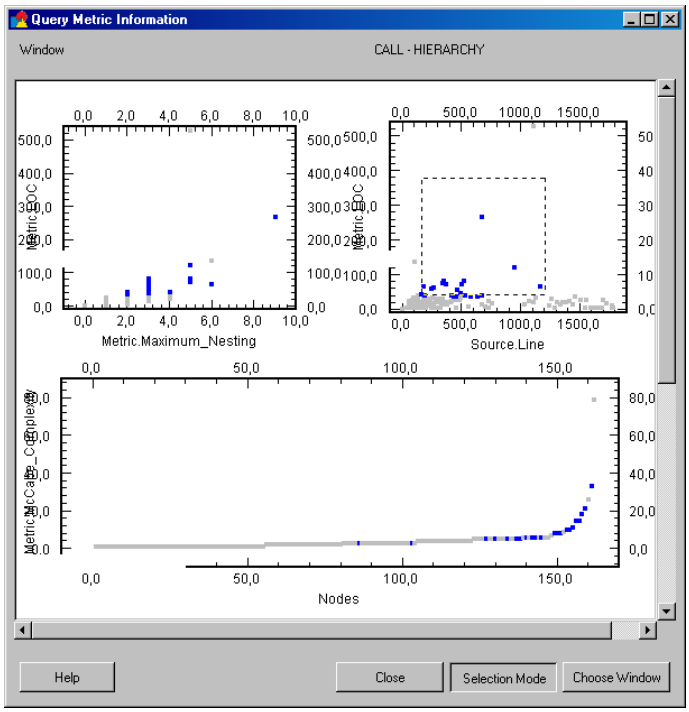


Abbildung 5.7.: Die Abbildung zeigt die Metric Box, mit deren Hilfe sich Metriken in Punkte-Diagrammen darstellen lassen.
(Quelle: Bauhaus Reference Guide)

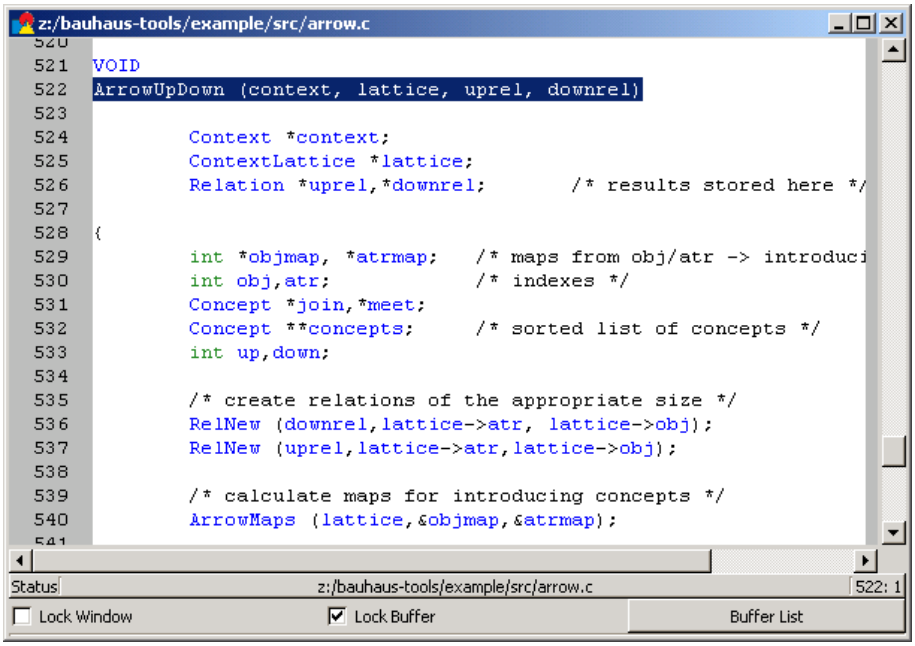


Abbildung 5.8.: Die Abbildung zeigt den Code Browser, mit dem Source Code innerhalb von GraVis betrachtet werden kann.
(Quelle: Bauhaus Reference Guide)

darstellt und im Folgenden als *Code Browser* bezeichnet wird (vgl. Abb. 5.1.3). Das betreffende Codefragment des korrespondierenden Knotens wird dabei farbig markiert und im aktuellen Codeausschnitt angezeigt.

Weitere Kompnenten Über das Menü der *Console* gelangt man zudem zu einigen Assistenten, mit denen Knoten und Kanten hinzugefügt oder entfernt werden können. Zudem finden sich hier weitere Tools und Dialoge, unter anderem zum Ausführen von Analysen und zum Importieren weiterer Sichten in der *Graph eXchange Language (GXL)*. Diese können insbesondere auch weitere Knoten und Kanten mit weiteren Attributen beinhalten.

5.1.4. Klonerkennungstools

Das Bauhaus-Projekt beinhaltet bereits drei separate Kommandozeilen-Tools zur Klonerkennung, deren Funktionsweise in den folgenden Abschnitten kurz erläutert wird. Dabei wird das Verfahren, die zugrunde liegende Granularität und Abstraktionsebene, sowie besondere Eigenschaften der verwendeten Technik erläutert. Zudem werden die verfügbaren Ausgabe-Formate genannt, die zuvor noch genauer erläutert werden sollen.

Clone Pair Format (CPF) – Ist ein Format mit zugehörigen *emacs-mode* [35], bei dem der Benutzer interaktiv durch die erkannten Klone browsen kann.

Graph eXchange Language (GXL) – Ist ein freies XML-basiertes Format zum Austausch von Graph-Informationen [39]. Das Format wurde zum Standard für Reverse-Engineering-Tools erklärt [36], wird aber auch in vielen anderen Bereichen verwendet. GraVis unterstützt den Import von Dateien dieses Formates (siehe Abschnitt 5.1.3).

Rigi Standard Format (RFS) – Ist ein Format des *Rigi graph editors*, der früher für Bauhaus verwendet wurde. GraVis unterstützt den Import von Graphen in diesem Format nur eingeschränkt[4].

Comma Separated Values (CSV) – Die Tools geben für den Import in ein Tabellenkalkulationsprogramm eine Zusammenfassung der Ergebnisse im CSV-Format an.

extended – Ist ein Format zur einfachen Weiterverarbeitung der Daten mit Kommandozeilentools, wie *cut* und *sed*.

[4] Im Fall der Klonerkennungswerkzeuge fehlen die Attribute an den importierten Kanten, die den Typ, sowie die Start- und End-Zeile des Klones in den jeweiligen Fragmenten angeben.

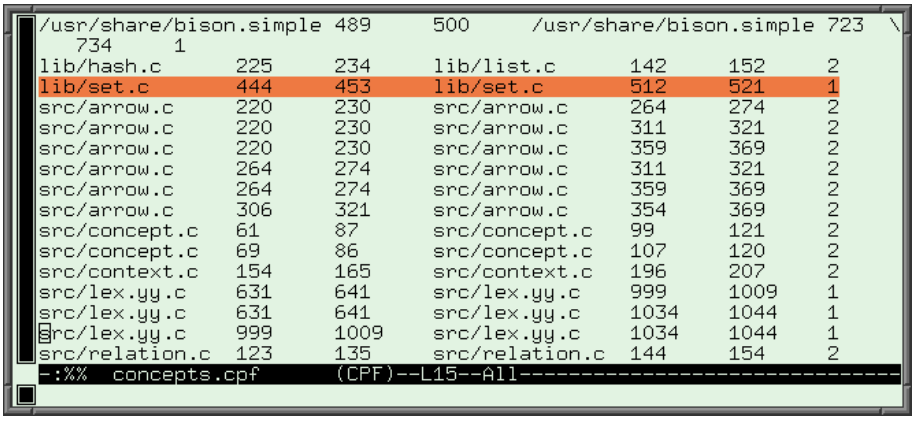


Abbildung 5.9.: Der Clone Pair Format (CPF) mode für Emacs.
(Quelle: Bauhaus Reference Guide)

ccdiml

Bei `ccdiml` handelt es sich um ein Kommandozeilen-Tool, das auf dem Verfahren von Ira D. Baxter beruht (siehe 2.2.4). Es vergleicht Teil-Bäume des abstrakten Syntax-Baumes der IML. Dementsprechend können nur Programmiersprachen unterstützt werden, für die das Generieren der IML bereits implementiert wurde.

Bei Verwendung der *Graph eXchange Language (GXL)* als Ausgabeformat kann das Ergebnis anschließend in GraVis importiert werden. Die Klone werden in diesem Fall als Kanten zwischen *Routine*-Knoten [5] dargestellt.

[5] Bei *Routine*-Knoten handelt es sich um Routinen oder Methoden

ccdiml	
Ansatz	Baxter [9]
Granularität	Anweisungsebene
Abstraktionsebene	AST, IML
erkannte Klone	Typ 1, Typ 2, Typ 3
unterstüzte Sprachen	C, C++
Ausgabeformate	CSV, CPF, RSF, GXL
In GraVis dargestellt als Kanten zwischen *Routine*-Knoten.	

clones

Das Tool `clones` beruht im Wesentlichen auf dem Ansatz von Brenda S. Baker (siehe 2.2.3) und wurde mit einigen Erweiterung versehen. So wurde unter anderem die Granularität auf die lexikalische Ebene angehoben und das Programm erkennt Typ 3-Klone. Da das Verfahren direkt auf dem Quellcode aufsetzt und nicht die IML als Eingabe voraussetzt, können neben Programmen in den Sprachen C, C++, Java und Ada auch in Cobol oder Visual Basic implementierte Anwendungen analysiert werden.

Bei Verwendung des GXL-Ausgabeformates für den GraVis-Import werden die Klone als Kanten zwischen *Module*-Knoten[6] repräsentiert.

[6] Ein *Module*-Knoten beschreibt in GraVis ein Modul oder die Kombination aus einer Quelldatei und zugehöriger *Header*-Datei.

clones

Ansatz	Baker [6, 7], Ukkonen [?]
Granularität	Token
Abstraktionsebene	lexikalisch
erkannte Klone	Typ 1, Typ 2, Typ 3
unterstüzte Sprachen	C, C++, Java, Ada, Cobol, Visual Basic
Ausgabeformate	extended, CSV, CPF, RSF, GXL
In GraVis dargestellt als Kanten zwischen *Module*-Knoten.	

cpdetector

Das Tool `cpdetector` befindet sich zur Zeit noch in der Entwicklung. Es basiert auf einem baker-ähnlichem Verfahren, bei dem jedoch zunächst der abstrakte Syntaxbaum, in diesem Fall in Form der IML bereits vorhanden, generiert, dieser wird anschließend wiederum *serialisiert* und dann in einen Suffix-Baum übertragen wird. Das Verfahren wird in einer Veröffentlichung, die in Kürze erscheinen soll, genauer beschrieben [11].

cpdetector

Ansatz	Baker [6, 7], Koschke et al. [11]
Granularität	Token
Abstraktionsebene	AST, IML
erkannte Klone	Typ 1, Typ 2, Typ 3
unterstüzte Sprachen	voraussichtlich C, C++, Java, Ada
Ausgabeformate	CSV, CPF, GXL, ...

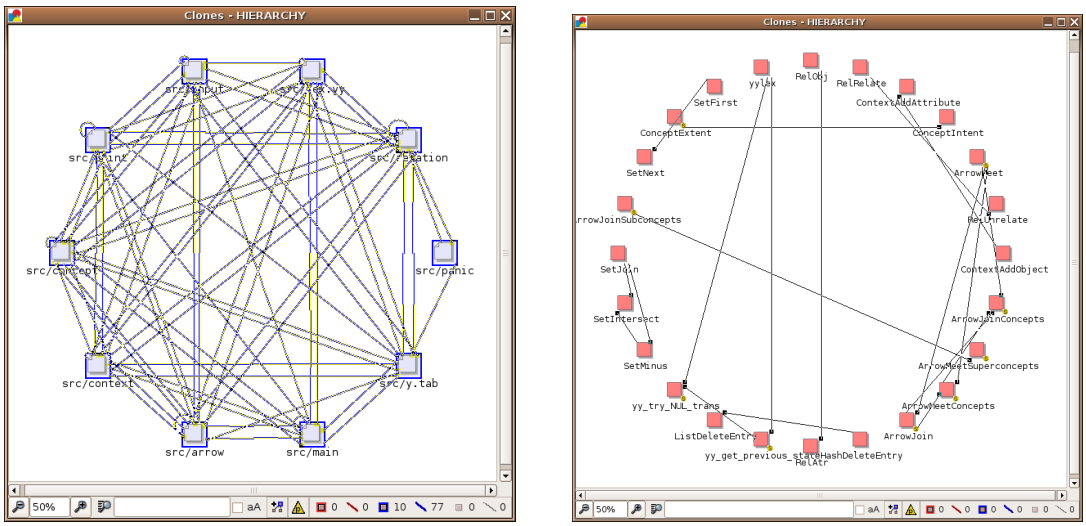


Abbildung 5.10.: Die Abbildung zeigt eine dem Duplication Web nicht unähnliche Darstellung der Kanten in einem Graph Window.

5.1.5. Klonanalyse

Die von den verschiedenen Tools erkannten Klone lassen sich, wie in den Beschreibungen erwähnt, über das GXL-Format in GraVis importieren und so bereits als Kanten zwischen Routinen, bzw. Modulen bewundern.

Über *Circular*-Layouts und die in GraVis integrierte Zoom-Funktionalität kann so bereits eine dem *duplication web* (siehe 4.1.2 auf Seite 37) nicht unähnliche Übersicht über die Klone erreicht werden[7].

Außerdem können über die Analyse *Lift-Edges (totally)* auf der *Module*-View die Kanten von der Routinen-, bzw. Modul-Ebene zur Verzeichnis-Ebene propagiert werden oder über die *Copy&Paste*-Funktion oder die *Union*-Analyse in andere Views übertragen, bzw. mit diesen vereint werden.

Zuletzt bietet das Bauhaus System mit dem oben erwähnten CPF-Mode für Emacs bereits eine leistungsstarke, jedoch ausschließlich textuelle und von GraVis kom-

[7] Es fehlen dabei allerdings die polymetrischen Darstellungen der Kanten-Dicke für die Anzahl der kopierten Codezeilen, sowie der Knoten-Breite für die Anzahl der intern kopierten Codezeilen.

plett entkoppelte Möglichkeit zur Analyse der von den Erkennungstools gelieferten Daten.

Im folgenden werden daher die verschiedenen Alternativen zur Integration von Visualisierungen zur explorativen Analyse der Ergebnisse von Klonerkennungstools untersucht und bewertet. Dazu werden im nächsten Abschnitt zunächst einige Vorüberlegungen beschrieben.

5.2. Vorüberlegungen

Wie bereits erwähnt richten sich die in Bauhaus integrierten Analysen und zugehörigen Werkzeuge nicht ausschließlich nur an Wartungsingenieure, sondern liefern ganz allgemein für die Softwareentwicklung und das Qualitätsmanagement interessante Ergebnisse. Dies trifft insbesondere auf eine Analyse bezüglich Code Duplikation zu.

Mit der Metric Box existiert bereits eine leistungsstarke und anwenderfreundliche Möglichkeit zur Analyse der verschiedenen metrischen Daten.

Diese ist prinzipiell auch zur Darstellung von Code Duplikation nutzbar. Dazu ist es lediglich erforderlich, die in Abschnitt 3.3 definierten Metriken zu implementieren.

Für das Qualitätsmanagement ist diese einfache und akkurate Veranschaulichung in Diagrammform zur Präsentation und Kommunikation der Ergebnisse weitgehend ausreichend. Dagegen ist für Entwickler und Wartungsingenieure eine explorative Analyse vom globalen Überblick zu punktuellen Informationen mit einer entsprechenden Benutzerinteraktion notwendig.

Der Entwickler ist dabei vor allem am eigenen Quellcode interessiert. Demgegenüber befasst sich der Wartungsingenieur eher mit Anpassungen am Gesamtsystem. Ihn dürfte dabei vor allem interessieren, wo ein bereits angepasstes Codefragmente durch Copy&Paste-Programmierung noch zu finden ist.

Festzustellen ist daher, dass die Analyse der Ergebnisse einer Klonerkennung alleine über die Metric Box, beziehungsweise die vorhandenen Visualisierungkonzepte in GraVis, bislang nicht ausreicht.

Bei der Erweiterung der vorhandenen Visualisierungskomponten oder der Integration neuer Konzepte müssen daher die unterschiedlichen Anforderungen dieser verschiedenen Benutzer berücksichtigt werden.

> *„Die Visualisierung [...] muss derart integriert werden, dass sie sich in das Gesamtkonzept, sowohl was die Interaktion als auch die Visualisierung betrifft, konsistent einfügt um den kognitiven Aufwand des Anwenders bei der Exploration und beim Verstehen der dargestellten Informationen zu minimieren.“* (Illes, 2004)

Zur Dimensionalität einer Darstellung stellt Illes in ihrer Arbeit [18] fest, dass die Frage einer zweidimensionalen oder dreidimensionalen Darstellung vom jeweiligen Kontext, in den die Visualisierung konsistent einzufügen ist und der Auswirkungen einer räumlichen Darstellung abhängt.

Im Falle einer räumlichen Darstellung, so Illes weiter, müsste der Anwender dreidimensionale und zweidimensionale Darstellungen gleichzeitig wahrnehmen (*Visualisierungskonsistenz*), die Navigation durch die jeweiligen Informationsräume würden sich zwangsläufig unterscheiden (*Interaktionskonsistenz*) und insbesondere durch den nicht intuitiv zu erkennenden Zusammenhang von räumlichen und zweidimensionalen Objekten erhöhe sich der *kognitive Aufwand*.

Sie kommt daher zu dem Schluss, dass eine räumlichen Darstellung im Falle von GraVis keine *Visualisierungskonsistenz*, keine *Interatkionskonsistenz*, sowie einen *erhöhten kognitiven Aufwand* zur Folge hätte.

Diese Einschätzung lässt sich generell auf den vorliegenden Fall übertragen.

Aus den genannten Gründen ist in diesem konkreten Fall einer zweidimensionalen Darstellung gegenüber einer dreidimensionalen der Vorzug zu gewähren.

5.3. Integrationsmöglichkeiten

Bei der Integration eines Konzeptes zur visuellen Analyse der Ergebnisse einer Softwareklonerkennung können grundsätzlich zwei Vorgehensweisen unterschieden werden. Einerseits können die bestehenden Komponenten um geeignete Techniken erweitert, andererseits vollkommen neue Konzepte integriert werden. In beiden Fällen sind die im Voraus definierten Ziele und Anforderungen an eine Visualisierung zu beachten (siehe Abschnitte 3.4). Insbesondere ist zu beachten, dass sich die Konzepte nahtlos in das vorhandene System einfügen und dem Benutzer so einen konsistenten Umgang mit diesem ermöglichen.

Nachfolgend wird die Integration der in Kap. 4 vorgestellten Visualisierungs-Konzepte untersucht und bewertet.

5.3.1. Polymetric Views

Das Graph Window bietet bereits ein Layout-Konzept, bei dem die Beziehungen zwischen den Knoten zu deren räumlichen Anordnung berücksichtigt werden kann (siehe 5.1.3). Dieses Konzept kann um die Berücksichtigung von metrischen Informationen, also der Attribute von Knoten und Kanten, erweitert werden. Über zusätzliche Abbildungen von Merkmalen auf weitere visuelle Attribute, wie Größe und Farbe von Knoten, sowie Kanten-Dicke würde die Darstellung beliebiger Polymetric Views ermöglicht. Unter anderem auch die in Abschnitt siehe 4.1.1 genannten Visualisierungen von Softwareklonen.

Ein Problem bei der Integration stellt das bisherige Konzept dar, in dem die unterschiedlichen Knotentypen bereits auf verschiedene Formen und Farben (siehe Abbildung 5.1.3 auf Seite 53) abgebildet werden. Zudem werden Kanten, abhängig von ihrem Typ, bereits eingefärbt.

5.3.2. Metric Box

Mit der Metric Box (siehe Abschnitt 5.1.3) existiert bereits ein leistungsstarkes Konzept zur Visualisierung von Metriken in GraVis. Durch die Implementation der in Abschnitt 3.3 erläuterten Metriken lassen sich diese mit Hilfe der Metric Box auch im Verhältnis zu bereits vorhandenen Metriken darstellen.

Zur Zeit unterstützt die Metric Box ausschließlich Punktdiagramme. Sie kann aber prinzipiell um beliebige Darstellungen erweitert werden [18]. Insbesondere die Implementation weiterer Diagrammtypen, wie Liniendiagramme, Stabdiagramme oder Histogramme ist in Erwägung zu ziehen.

5.3.3. Code Browser

Der bereits integrierte Code Browser (siehe Abschnitt 5.1.3) stellt auf Wunsch textuell die verschiedenen Quelldateien dar. Das zum jeweils gewählten Knoten oder zur jeweils gewählten Kante gehörende Codefragment wird dabei über eine farbige Markierung hervorgehoben. Diese könnte auch zur Darstellung der einem Klon zu Grunde liegenden Quellfragmente genutzt werden. Über eine zusätzliche *Dekoration* könnten weitere Informationen, die jeweilige Zeile betreffend, mit direktem Raumbezug (siehe Abschnitt 3.4.4) angezeigt werden.

5.3.4. TreeMap

Mit Hilfe von Tree Maps kann intuitiv die Ausprägungen eines Merkmals, wie den *Lines of Copied Code (LCC)*, eines Software-Systems überblickt werden (siehe Abschnit 4.1.6 auf Seite 43). Über die zusätzliche Einfärbung der Rechteckflächen der Knoten der tiefsten Ebene kann eine weitere Metrike, wie ein Verhältnis von intern (LIC) zu extern (LEC) kopierter Codezeilen visualisiert werden. Über einen geeignete Interaktionsmechanismus können sich explorativ die Daten der oberen bis zur elementaren Stufe betrachtet werden und so alle Zielklassen einer Visualisierung abgedeckt werden.

5.3.5. Bewertung

Die Erweiterung des Graph Windows um Polymetric Views und der Metric Box um weitere Diagrammtypen bieten beide den Vorteil des bereits integrierten Benutzungskonzeptes.

Die Polymetric Views, sowie die Tree Map decken zudem alle Zielklassen einer Visualisierung ab und eignen sich in Verbindung mit dem vorhandenen Graph Window daher gegenüber den anderen Konzepten besonders zur explorativen Analyse der Daten.

Die Probleme der Polymetric Views können über eine Auswahl der darzustellenden Metriken und visuellen Variablen umgangen werden. So könnten verschiedene Layout-Algorithmen auf Wunsch angewendet werden, die dabei unterschiedliche Metriken darstellen.

Die Tree Map bietet demgegenüber den Vorteil einer kompakten Darstellung, die die Ausprägungen von Merkmalen zu einem bestimmten Zweck veranschaulicht. Daher lassen sich aus dieser mit geringem kognitiven Aufwand intuitiv auffallende Werte erkennen. Zusätzlich kann sie auf leichte Weise zur Darstellung anderer Metriken verwendet werden.

Die Dekoration im Editor kann als zuschaltbares PlugIn realisiert werden und eignet sich, wie der DotPlot, nur zur Darstellung von „Details auf Wunsch“ zu einzelnen Beobachtungspunkte des Graph Windows oder der Tree Map. Neben punktuellen Informationen sind beide dazu im Stande auch lokale Informationen anzuzeigen. Die Dekoration im Editor hat dabei den Vorteil des direkten Raumbezugs gegenüber dem Dot Plot, der in einem eigenen Fenster integriert werden müsste. Über eine optional einblendbare Dekoration können zudem nahezu beliebig viele verschiedene punktuelle Informationen angezeigt werden.

5.3.6. Fazit

Nach abwägen der aufgeführten Vor- und Nachteile wurde die Kombination aus einer Tree Map mit der optional zuschaltbaren Dekoration im Code Browser als geeignetes

Visualisierungskonzept zur Analyse von Code Duplication gewählt.

Über die Tree Map kann der Benutzer adäquat Komponenten des Systems mit hoher Belastung an dupliziertem Code identifizieren und über die Dekoration im Code Browser werden ihm witere Detail-Informationen zu den Klonen der gerade betrachteten Datei geboten. Über die ohnehin für die ebenfalls metrikbasierte Tree Map nötige Implementation der in Abschnitt 3.3 erwähnten Metriken steht mit der Metric Box eine weitere Visualisierung zu Präsentations- und Kommunikationszwecken zur Verfügung.

5.4. Konzept

Hier wird nun das gewählte Visualisierungskonzept einer *Tree Map View* in Verbindung mit der zuschaltbaren *Dekoration* im Code Browser erläutert.

Zunächst wird das verwendete Konzept der Tree Map View näher betrachtet und die Integration in das bestehende Konzept, sowie das Benutzerinteraktiondesign genauer beschrieben. Darauf folgt eine Erläuterung der Dekoration im Code Browser. Zum besseren Verständnis ist das ganze mit (hoffentlich) ausreichend Bildern angereichert.

Der letzte Abschnitt befasst sich mit der Bewertung hinsichtlich der gesetzten Ziele. Zum Einen ist eine adäquate Visualisierung nahtlos in das bestehende Konzept zu integrieren, zum Anderen sind die Bedürfnisse und Anforderungen der verschiedenen potentiellen Bauhaus-Nutzer zu erfüllen.

5.4.1. Tree Map View

Die aktuelle Visualisierungsform der *Tree Map View* beinhaltet neben der eigentlichen *Tree Map* eine optional einblendbare *Tree List* samt Address- und Status-Zeile.

Die Tree Map selbst bildet dabei die Lines of Copied Code (LCC) eines Knotens im RFG auf die Größe der jeweiligen Fläche und das Verhältnis von intern kopierter

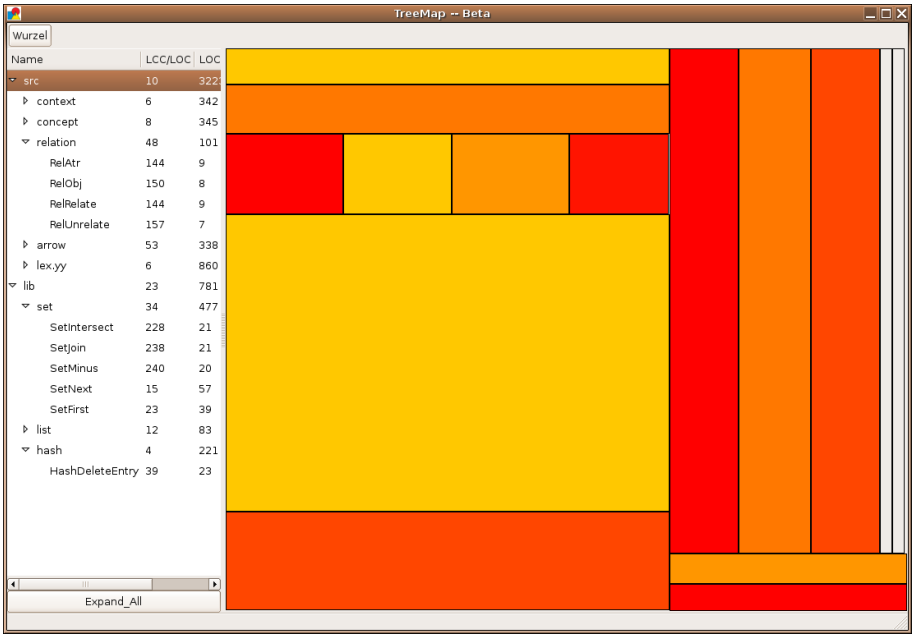


Abbildung 5.11.: Das Beispiel zeigt eine Vorabversion der Tree Map View.

Zeilen zu extern kopierter Zeilen auf die Farbe der Fläche ab. Je höher dabei der Anteil der intern kopierten Zeilen ausfällt, um so roter, je geringer um so gelber ist die jeweilige Fläche[8] eingefärbt.

An der linken Seite neben der Tree Map kann eine List View eingeblendet werden, die weitere Informationen, wie Name und Werte der Knoten für die Metriken LOC, LIC und LEC darstellt (vgl. Abschnitt **??**). Beide können auf einfache Weise um die Darstellung zusätzlicher Metriken erweitert werden.

Die Integration der Tree Map View in GraVis erfolgt in Form eines eigenständigen Dialogs unter Ausnutzung des vorhandenen Mechanismus zur Benutzerinteraktion. Dadurch wird dem Benutzer ein einheitliches Benutzungskonzept geboten.

Eine Tree Map View bezieht sich dabei immer auf das Graph Window, aus dem

[8]Im folgenden wird die einem Knoten im RFG zugeordnete Fläche einer Tree Map direkt als Knoten bezeichnet.

sie gestartet wurde. Sie stellt dementsprechend nur sichtbare Knoten des jeweiligen RFG-Teilgraphen dar, der in dem jeweiligen Graph Window angezeigt wird. Über den Button „Choose Window“ in der Statusleiste des Fensters kann der jeweilige Map View eine anderes Graph Window und damit eine andere RFG View zugeordnet werden. Da sich beliebig viele Graph Windows pro RFG View gleichzeitig öffnen lassen, können mehrere Map Views die gleiche RFG View darstellen.

Navigation

Mit der linken Maustaste kann der aktuell unter dem Mauszeiger befindliche Knoten in der Tree Map ausgewählt werden. Mit einem Klick auf die rechte Maustaste wird ein Kontextmenü zum Vorschein gebracht, in dem verschiedene Operationen den jeweiligen Knoten betreffend zur Verfügung stehen. Die Operationen der ersten Kategorie haben dabei ausschließlich Auswirkungen auf die Visualisierung der aktuellen Tree Map View. Über diese kann der jeweilige Knoten expandiert, kollabiert oder auf die volle Größe der Tree Map „gezoomt“ werden. In der zweiten Kategorie befinden sich die *Selektion* betreffende Operationen, deren Funktionsweise im folgenden Abschnitt beschrieben wird. Zuletzt findet sich ein Eintrag zum Anzeigen des Quellcodes des zugehörigen RFG-Knotens.

Die Operationen der ersten Kategorie beeinflussen neben der Tree Map selbst auch die Darstellung der Daten in der optional einblendbare Tree List an der linken Seite. Dabei handelt es sich um eine Listen-Ansicht, die in einer Baum-Struktur Informationen zu den aktuell expandierten Knoten anzeigt. Wird ein Knoten über das Kontextmenü der Tree Map expandiert oder kollabiert, so geschieht dies gleichzeitig auch in der Tree List. Gleichzeitig kann jeder Knoten auch über die Tree List expandiert und kollabiert werden. Dieses wirkt sich ebenfalls auf die Darstellung in der Tree Map aus. Die Tree List zeigt neben dem Namen des Knotens die Werte einiger Metriken an[9]. An ihrem unteren Rand befindet sich ein Button „Expand All“ mit dem alle Knoten der Tree Map View rekursiv expandiert werden.

[9] Wie bereits erwähnt, handelt es sich dabei um die LOC, LCC, LEC, LIC, sowie ein Verhältnis von LCC zu LOC.

Die Status-Zeile am unteren Bildschirmrand zeigt den Namen des aktuell unter dem Mauszeiger befindlichen Knotens an, sowie den Namen der RFG View des aktuell gewählten Graph Windows. In der Address-Leiste am oberen Rand des Fensters wird den aktuell gewählten Wurzel-Knoten der Tree Map sowie dessen Väter-Knoten angezeigt. Über die bereits erwähnte „Zoom“-Operation des Kontextmenüs der Tree Map, wird der aktuell unter dem Cursor befindliche Knoten zum Wurzel-Knoten und auf die volle Größe der Tree Map „gezoomt“. Dies hat jedoch keine Auswirkung auf die Tree List. Über die Address-Leiste kann auch bei ausgeblendeter Tree List zu einem in der Hierachie höher liegendem Knoten zurück gewechselt werden.

Selektion

Über das Kontextmenü und die linke Maustaste kann ein Knoten beziehungsweise eine Fläche in der Tree Map *selektiert* werden. Diese wird dann grafisch mit einer Umrandung hervorgehoben. Eine Mehrfachselektion verschiedener Flächen ist auch möglich. Die entsprechenden Einträge in der Tree List werden ebenfalls selektiert und blau hinterlegt. In dem Graph Window, auf das sich die Tree Map View bezieht, wird der korrespondierende Knoten gleichfalls selektiert[10]. Umgekehrt wird bei der Selektion eines Knotens im Graph Window die entsprechende Fläche in der Map View hervorgehoben.

Filter

Neben der bereits erwähnten Filter-Funktionalität – über das Kontextmenü der Tree Map oder die Tree List Knoten nach Wunsch ein- und ausblenden zu können – bietet die Tree List außerdem die Möglichkeit nach einem Knoten zu suchen. Über das jeweilige Graph Window können zudem Knoten ausgeblendet werden, so dass sie auch in der Tree Map nicht mehr angezeigt werden. Diese Funktionalität sollte aber mit Vorsicht verwendet werden, da hierdurch ein falscher Eindruck beim Betrachten der

[10] Im Graph Window werden selektierte Knoten mit einem blauen Rahmen dargestellt.

daraus resultierenden Map entstehen kann. So wird beim Expandieren eines Knotens die verfügbare Fläche unter den Kind-Knoten, abhängig von der Ausprägung des jeweiligen Merkmals, in diesem Fall der kopierten Code-Zeilen, aufgeteilt. Wird nun einer oder mehrere dieser Knoten ausgeblendet, kann es unter Umständen dazu kommen, dass ein Knoten mit geringer Ausprägung des jeweiligen Merkmals einen Großteil der Fläche zugeteilt bekommt.

Interaktionsdesign

Durch die Konzepte der Navigation innerhalb der Tree Map und die Selektion und damit Interaktion mit den anderen Visualisierungskomponenten in GraVis wurde ein geeignetes Konzept zum effektiven Interaktionsdesign (vgl. Abschnitt 3.4.2 auf Seite 31) in die Tree Map View integriert.

Dem Benutzer ist es demnach möglich zunächst einen Gesamteindruck über die Ausprägung der Klone in den über die jeweilige RFG View gewählten Knoten zu erhalten. Mit Hilfe der vorgestellten Navigationsmittel der Tree Map und der Möglichkeit der Selektion zwischen den verschiedenen Fenstern können interessante Teilmengen genauer betrachtet werden. Durch die Selektion eines Knotens können in der Tree List Informationen zu diesem abgelesen werden. Man erhält als punktuelle Detailinformationen Name oder Metrikwerte. Über das Kontextmenü kann zudem der Quelltext des jeweiligen Knotens betrachtet werden.

5.4.2. Dekoration im Editor

Um den Anwender nach der Identifikation eines konkreten Knotens, also beispielweise einer Funktion oder Datei, weitere Informationen bieten zu können, wurde der Editor um eine Dekoration erweitert. Auf der linken Seite des Editors befinden sich Detailinformationen zu den Klonen der gerade betrachteten Codezeilen, auf der rechten Seite zu den Klonen der gesamten Datei. Die Dekoration wurde in Form eines optional einblendbaren Plugins in das bestehende Konzept des Editors integriert.

Ist eine Quellcodezeile Teil eines Klonfragmentes so wird jetzt an ihrer linken Seite ein farbiges Rechteck angezeigt. Die Farbe ist eindeutig durch das Klon-Cluster des Codefragmentes bestimmt. Diese Einfärbung hat den Vorteil, dass ähnliche Codezeilen aufgrund gleicher Farbe erkannt werden können.

Auf der rechten Seite wird jetzt neu neben dem vertikalen Scroll-Balken zusätzlich eine pixelbasierte Übersicht über die Klonfragmente innerhalb der Datei angezeigt. Dabei wird auf diesen senkrechten Balken jede Codezeile der Datei in gleichbleibender Reihenfolge als eine Pixelzeile abgebildet. Handelt es sich bei der betreffenden Codezeile um einen Klon, so wird dessen Pixelzeile in dem Balken entsprechend eingefärbt, andernfalls bleibt sie grau. Der Pixelbalken wird dabei immer auf die volle Größe skaliert. So entspricht die erste Pixelzeile am oberen Rand des Fensters der ersten Zeile der Datei und die letzte Pixelzeile, die sich immer am unteren Rand des Fensters befindet, der letzten Codezeile der Datei. Im Extremfall kann so bei einer einzeiligen Datei der gesamte Balken aus einer Pixelzeile bestehen und wäre somit komplett in der Farbe der einen Zeile eingefärbt[11]. Gleichzeitig fallen bei größeren Dateien, die jeweiligen Pixelzeilen entsprechend kleiner aus. Die Einfärbung der einzelnen Pixelzeilen erfolgt dabei wiederum nach der Farbe des jeweiligen Klon-Clusters.

Interaktionsdesign

Im Gegensatz zur Tree Map bietet die Dekoration nur ein stark vereinfachtes Interaktionsdesign. Nach einem Klick auf eines der Kasten-Symbole am linken Rand wird ein größeres Fenster links neben der Zeilennummerierung innerhalb des Editor-Fensters eingeblendet. Dieses stellt nun in einer Art Datei-Codezeilen-Matrix die Klone in anderen Dateien mit der ausgewählten Codezeile bzw. dem ausgewählten Codefragment dar. Das Fenster besteht dabei aus mehreren Spalten, die jeweils eine Datei repräsentieren. In diesen Spalten wird wiederum ein rechteckiges Symbol in der jeweiligen Zeile angezeigt, wenn diese auch im Quellcode der entsprechenden Datei zu finden ist. Die Farbe der Symbole bleibt dabei per Definition der Klon-Cluster iden-

[11] Was in einem solchen Fall wohl keine, also grau, wäre.

tisch[12]. Klickt der Anwender nun mit der linken Maustaste auf eines dieser gerade eingeblendeten Kasten-Symbole, so wird in einem zweiten Editorfenster die entsprechende Datei geöffnet und die betreffende Codezeile angezeigt.

Ähnlich verhält es sich auf der rechten Seite. Bei einem Klick mit der linken Maustaste auf eine der Pixelzeilen werden spaltenweise alle Dateien mit denen die jeweilige Datei einen Klon hat auf gleiche Weise in Form von Pixelzeilen dargestellt. So erhält der Anwender an Hand der Farben einen schnellen Überblick, in welchen Dateien sich welche der Codefragmente als Duplikate noch verbergen. Mit einem Klick auf eine der Pixelzeilen der eingeblendeten Spalten wird wiederum ein weiteres Editorfenster mit der jeweiligen Datei an der entsprechenden Position geöffnet.

Um bereits im Voraus zu sehen, um welche Datei es sich handelt, wird nach kurzer Verweildauer über einer der Spalten – auf der linken, wie rechten Seite – ein so genannter Tooltip mit dem Namen der zu der jeweiligen Spalte gehörenden Datei eingeblendet.

5.4.3. Anwendungsszenarien

Im Folgenden soll nun geprüft werden, ob die gewählten Visualisierungen die unterschiedlichen Bedürfnisse und Anforderungen der verschiedenen Anwender des Bauhaus Systems erfüllen. Das Projekt lässt sich, wie bereits erwähnt, neben Reengineering- auch zu Engineering-Zwecke und im Qualitätsmanagement einsetzen. Die Zielgruppe bilden daher Personen aus diesen drei Bereichen. Für jede dieser Gruppen wurde daher ein typischer Anwendungsfall gewählt, der im Folgenden detailliert beschrieben und dessen Umsetzung mit Hilfe des vorgestellten Konzepts abschließend bewertet wird.

[12] Obwohl die Farbe durch das Klon-Cluster der jeweiligen Zeile bzw. des jeweiligen Codefragmentes bestimmt wird, werden in den Spalten nur die Klone der Klonklasse, in der sich die jeweilige Zeile bzw. Codefragment befindet angezeigt (vgl. Abschnit 2.1.5).

Szenario 1: Wartungsarbeiten

Das System richtet sich in erster Linie an Wartungsingenieure. Sie können mit Hilfe dessen Architektursichten rekonstruieren und verschiedene Aspekte eines Altsystems analysieren. Wartungsingenieure stehen dabei oft in der Pflicht, ohne Vorwissen Anpassungen an einem solchen Legacy System vornehmen zu müssen.

Der erste Anwendungsfall umfasst genau dieses Szenario, in dem der Wartungsprogrammierer zur Aufgabe hat, eine gegebene Funktionalität des Systems zu refaktorisieren.

Zunächst überführt er dazu den Source-Code des jeweiligen Systems mit dem entsprechenden Sprachen-Frontend (siehe Abschnitt 5.1.1) in die RFG-Darstellung. Mit den bereits integrierten Analysen und Werkzeugen identifiziert er die entsprechende Funktionalität. Nachdem er diese an die neue Anforderung angepasst hat, ist er natürlich an weiteren duplizierten Vorkommen dieser Funktionalität interessiert[13].

In der Regel ist dabei nicht dokumentiert, an welche Stellen eine Funktionalität kopiert wurde. Bisher blieb dem Programmierer somit keine andere Wahl, als mühsam nach weiteren Vorkommen der Funktionalität zu suchen und über *Trial&Error* festzustellen, ob nun alle Vorkommen auf die neue Anforderung angepasst wurden, ohne sicher sein zu können, nicht doch eine Stelle „übersehen“ zu haben.

Mit dem neu hinzugekommenen Konzept der zuschaltbaren Dekoration in Form eines Plugins für den Code Browser erhält er nun nach einem Klick auf die Dekoration an der linken Seite in der betreffenden Zeile einen Überblick über alle weiteren Vorkommen des jeweiligen Codefragmentes. Über die Dekoration auf der rechten Seite ist es ihm zudem möglich Informationen über die Position der Fragmente in den anderen Dateien, sowie den Zusammenhang mit weiteren Klonen zu erkennen.

Dazu wählt er einfach beispielsweise über die Suchfunktion des Graph Window[14] oder der Tree Map, den Knoten der entsprechenden Datei und lässt sich über das

[13] Alternativ ist er auch zunächst an den weiteren Vorkommen interessiert und überarbeitet dann die jeweilige Funktionalität – das spielt hier aber keine Rolle.

[14] vorzugsweise in der `File View`

Kontextmenü dessen Source Code im Code Browser anzeigen. Voraussetzung ist dabei allerdings, dass er zuvor eine der Klonerkennungstools verwendet und die entsprechend erstellte GXL-Datei mit den Klon-Kanten in GraVis importiert hat. Durch das Öffnen einer Tree Map View werden dann automatisch alle benötigten Analysen zur Berechnung der Metriken, *Liften* der Kanten sowie das Laden des Plugins ausgeführt.

Szenario 2: Code-Analyse

Das zweite Szenario beschäftigt sich konkreter mit einer Anaylse der Ergebnisse der Erkennungstools. Um die negativen Auswirkungen von Code Duplikationen und anderen *bad smells* zu verhindern, ist es nötig ein System von Zeit zu Zeit diesbezüglich zu untersuchen und ggf. erkannte Klone zu beheben.

Während der *Forward Engineer* dabei vor allem an möglichen bad smells und damit Duplikationen innerhalb seiner Dateien, Module oder Pakete interessiert ist, hat der Wartungsingenieur eher allgemeiner das system-weite Untersuchen des gesamten Quelltextes auf mögliche Duplikation zum Ziel.

Das Vorgehen ist aber in beiden Fällen ähnlich. Zunächst muss der Quellcode des Systems in die RFG-Darstellung überführt und eins der Klonerkennungstools angewendet werden. Das Ergebnis wird in Form einer GXL-Datei als View in GraVis importiert. Daraufhin wählen sie über das bestehende Konzept der RFG Views und der Graph Windows die sie interessierenden Knoten[15]. Über das Kontext-Menü des jeweligen Graph Windows kann dann die Tree Map View geöffnet werden. Über deren zuvor erläuterte Konzepte zur effektiven Benutzerinteratkion in Verbindung mit den Interaktionsmechanismen des Graph Window können sie explorative die Klon-Situation analysieren. Bei interessanten Knoten erhalten sie auf Wunsch über den Code Browser Detail-Informationen zu dessen Klone und können über die Dekoration die weiteren Vorkommen untersuchen und so den jeweiligen Klon ggf. an allen in Frage kommenden Stellen leicht ersetzen.

[15] Da die Tree Map View die physikalische und logische Dekomposition eines Systems berücksichtigt, empfiehlt es sich zur Analyse ein Graph Window zu wählen – bzw. eine RFG View zu wählen oder über die in Abschnitt XXX erwähnte Analyse-Funktionen zu erstellen – das auch Knoten dieser Typen beinhaltet.

Den bisherigen Benutzern des CPF-Mode für Emacs wird mit der optional einblendbaren Tree List zudem ein konsistenter Übergang in das neue Konzept gewährt. Diese stellt in ähnlicher Weise die Werte verschiedener Metriken dar[16], ist aber gegenüber dem Emacs-Mode voll in das Konzept zur Benutzerinteraktion der Tree Map View und damit von GraVis integriert.

Szenario 3: Qualitätsmanagement

Der dritte Anwendungsfall schließlich befasst sich mit den Anforderungen des Qualitätsmanagements. Ähnlich dem zweiten Szenario steht dabei die Analyse bestimmter Eigenschaften eines Systems, in diesem Falle die Duplikat-Situation eines Software-Systems, im Vordergrund. Das Interesse liegt dabei aber nicht in einer Analyse mit dem Ziel, konkrete Klone zu erkennen und ggf. zu beseitigen, sondern allgemeiner in der Bewertung der Belastung und damit der Qualität des Systems insbesondere im Verhältnis oder Zusammenhang mit anderen Metriken. Schwerpunkt ist dabei vor allem, die Ergebnisse zu Kommunikations- und Präsentationszwecken geeignet darzustellen.

Mit der Metric Box ist bereits ein Tool in GraVis integriert, dass dieser Aufgabe mehr als gerecht wird. Mit der erwähnten Implementation der in Abschnitt 3.3 auf Seite 27 beschriebenen Metriken, eignet sich diese auch zur angesprochenen Analyse von Duplikation innerhalb eines Systems.

[16] Die Tree List lässt sich auf einfache Weise um die Darstellung weiterer Spalten mit zusätzlichen oder anderen Metriken erweitern.

Kapitel 6.

Zusammenfassung und Ausblick

6.1. Bewertung

Das vorgestellte Konzept bietet adäquate Visualisierungen zur Analyse der Ergebnisse einer Software-Klonerkennung. Mit den vorgestellten Mechanismen zur Selektion und Projektion sowie den Details auf Wunsch erfüllt es alle Ziele einer Softwarevisualisierung.

Mit der Verwendung der in GraVis bereits integrierten Mechanismen zur Benutzerinteraktion erfüllt es zudem die gestellte Anforderung einer nahtlosen Integration in das vorhandene Konzept und bietet in Verbindung mit dem Navigationkonzept innerhalb der Tree Map eine effiziente Möglichkeit zur explorativen Analyse der Daten.

Wie die Anwendungsszenarien aus Kapitel 5.4.3 zeigen, erfüllt es außerdem die grundlegenden Anforderungen und Bedürfnisse der verschiedenen Anwender aus der Zielgruppe des Bauhaus-Projektes.

Mit der in die Tree Map View integrierten Tree List wird Benutzern, die bisher den CPF-mode für Emacs verwendet haben, ein ähnliche Form der Darstellung zur Verfügung gestellt, die allerdings komplett in die GraVis-Konzepte zur Benutzerinteraktion integriert ist.

Über die neu integrierten Metriken wurde zudem die bereits vorhandenen Metric Box um die Möglichkeit zur Auswertung und Präsentation der Daten einer Klonerkennung erweitert.

Die definierten Ziele einer adäquaten Visualisierung mit effektivem Interaktionsdesign und der Unterstützung der verschiedenen Anwender wurden damit erreicht.

6.2. Erweiterungen

Aufsetzend auf das vorgestellten Konzept, sind verschiedene Erweiterungen denkbar, die dem Benutzer zusätzliche Funktionen oder völlig neue Konzepte bieten. Diese sollen im Folgenden kurz angesprochen werden.

6.2.1. Tree Map View

Das Konzept der Tree Map View basiert auf metrischen Daten[1] (vgl. Kapitel 4.1.6). Sie eignet sich deshalb prinzipiell zur hierarchischen Darstellung beliebiger Eigenschaften eines Software-Systems. Über ein Konfigurationsdialog könnte eine flexible und anwenderfreundliche Möglichkeit geschaffen werden, die aktuell dargestellten Metriken sowohl in der Tree Map als auch in der Tree List einzustellen. Zudem kann die Tree Map um die Abbildung weiterer Merkmale auf zusätzliche visuelle Variablen, wie in Kapitel 4.1.6 beschrieben, erweitert werden.

6.2.2. Code Browser

Eine Unzulänglichkeit bei dem vorgestellten Konzept, stellt die nicht vorhandene Editor-Funktionalität des Code Browsers dar. So können über dessen neue Dekoration Kopien von Funktionalitäten gefunden und betrachtet, jedoch nicht behoben werden. Dieses muss der Anwender immer noch über ein externes Tool erledigen. Die Möglichkeit der Modifikation des aktuell untersuchten Quelltextes über den Code Browser hätte allerdings zur Folge, dass die gesamten getätigten Analysen damit

[1] Im Fall von GraVis in Form von Attributen an den Knoten.

automatisch ungültig werden würden. Im Fall der Klonerkennung würden nach dem Entfernen einer Zeile beispielsweise alle Zeilenangaben, die sich auf Folgezeilen beziehen, inkorrekt.

6.2.3. Andere Erweiterungen

Die Integration neuer Diagrammtypen in die Metric Box stellt, wie die Integration des bereits erwähnten Dot Plots eine weitere interessante Möglichkeit zur Erweiterung dar. Letzterer ist aber auf Grund seines speziellen Verwendungszwecks zur Darstellung der Gemeinsamkeiten und Unterschiede zweier Dateien weniger interessant.

6.2.4. Evolutionäre Aspekte

Für das Qualitätsmangament ist es äußerst interessant, die Duplikat-Situation eines Software-Systems über einen längeren Zeitraum zu beobachten. Zu diesem Zweck existieren verschiedene Visualisierung, die metrische Informationen im Verlaufe des Lebenszyklus eines Systems visualisieren.

6.3. Persönliches Fazit

Leider konnte die Integration der vorgestellte Tools noch nicht vollständig abgeschlossen werden. Unter anderem aus diesem Grund fehlt auch ein entsprechendes Kapitel die konkrete Integration betreffend in dieser Arbeit.

Die Entwicklung eines weiteren Tools wurde auf Grund von Problemen im Zusammenhang mit GtkAda eingestellt. Dabei handelt es sich um die so genannte *Klon Matrix*, die ähnlich wie die Tree Map eine Navigation über die Knoten deren hierarchischen Struktur entsprechend erlaubt. Die Knoten einer Hierarchie-Ebene werden dabei auf die Zeilen, bzw. Spalten einer *Matrix* aufgetragen. Knoten in der Matrix repräsentieren die Klon-Kanten eines RFG View zwischen den jeweiligen RFG

Knoten. Verschiedene Eigenschaften der RFG Knoten und Kanten können über Metriken auf die visuellen Variable, wie Größe und Farbe der Knoten in der Matrix View abgebildet werden.

Ein Vergleich verschiedener Visualisierungstools von Softwaremetriken und insbesondere Tools zur Klonvisualisierung konnte aus zeitlichen Gründen leider nicht mehr in dieses Dokument integriert werden.

Kapitel 7.

Literaturverzeichnis

7.1. Software-Klone

[1] Bellon, S.: *Vergleich von Techniken zur Erkennung duplizierten Quellcodes*. Diplomarbeit, Universität Stuttgart, 2002.

[2] Fowler, M. und K. Beck: S. Chapter 3: Bad Smells in Code. Addison Wesley, July 1999.

[3] Koschke, R.: *Software-Reengineering*. Universität Bremen, 2006. Skript zur Vorlesung.

[4] Rieger, M.: *Effective Clone Detection*. Doktorarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern, 2005.

7.2. Klonerkennung

[5] Baker, B.: *Parameterized Diff*. In: *ACM-SIAM Symp. on Discrete Algorithms*, S. S854–S855, Jan. 1999.

[6] Baker, B. S.: *A Program for Identifying Duplicated Code*. Computing Science and Statistics, 24:49–57, 1992.

[7] Baker, B. S.: *On Finding Duplication and Near-Duplication in Large Software Systems*. In: Wills, L., P. Newcomb und E. Chikofsky (Hrsg.): *Second*

Working Conference on Reverse Engineering (WCRE), S. 86–95. IEEE Computer Society Press, July 1995.

[8] Baker, B. S. und R. Giancarlo: *Longest Common Subsequence from Fragments via Sparse Dynamic Programming*. In: *European Symposium on Algorithms*, S. 79–90, August 1998.

[9] Baxter, I. D., A. Yahin, L. Moura, M. Sant'Anna und L. Bier: *Clone Dectection Using Abstract Syntax Trees*. In: *Proceedings; International Conference Software Maintenance (ICSM)*, S. 368–377, 1998.

[10] Ducasse, S., M. Rieger und S. Demeyer: *A Language Independent Approach for Detecting Duplicated Code*. In: Yang, H. und L. White (Hrsg.): *Proceedings ICSM'99 (International Conference on Software Maintenance)*, S. 109–118. IEEE, 1999.

[11] Koschke, R., R. Falke und P. Frenzel: *Clone Detection Using Abstract Syntax Suffix Trees*. to be appear, 2006.

[12] Krinke, J.: *Identifying Similar Code with Program Dependence Graphs*. In: *Proc. Eigth Working Conference on Reverse Engineering*, S. 301–309, 2001.

[13] Mayrand, J., C. Leblang und E. M. Merlo: *Experiment on the Automatic Detection of Function Clones in a Software System using Metrics*. In: *Proceedings of the International Conference on Software Maintenance (ICSM)*, S. 244–254, 1996.

7.3. (Klon-)Visualisierungen

[14]

[15] Bertin, J.: *Graphische Darstellungen und die graphische Weiterverarbeitung der Informationen*. de Gruyter, 1982.

[16] Dässler, R. und H. Palm: *Virtuelle Informationsräume mit VRML*. dpunkt Verlag, 1998.

[17] Holten, D., R. Vliegen und J. J. van Wijk: *Visual Realism for the Visualization of Software Metrics*. In: *3rd IEEE Workshop on Visualizing Software for Understanding and Analysis*, S. 27–32, 2005.

[18] Illes, T.-M.: *Visualisierung von Metriken mit GraVis*. Diplomarbeit, Universität Stuttgart, 2004.

[19] Kasper, H.: *Redesign von Benutzungsoberflächen durch Mittel der Navigation*. Diplomarbeit, Universität Stuttgart, 2003.

[20] Lanza, M. und S. Ducasse: *Polymetric views — a lightweight visual approach to reverse engineering*. In: *IEEE Transactions on Software Engineering*, S. 782–795, 2003.

[21] Mackinlay, J.: *Automating the design of graphical presentations of relational information*. ACM Trans. Graph., 5(2):110–141, 1986.

[22] Rieger, M. und S. Ducasse: *Visual Detection of Duplicated Code*. In: Ducasse, S. und J. Weisbrod (Hrsg.): *Proceedings ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, Nr. 6/7/98. Forschungszentrum Informatik Karlsruhe, 1998.

[23] Rieger, M., S. Ducasse und M. Lanza: *Insights into System-Wide Code Duplication*. 2004.

[24] Robertson, P. K.: *A methodology for scientific data visualization: Choosing representations based on a natural scene paradigm.*. October.

[25] Schumann, H. und W. Müller: *Visualisierung. Grundlagen und allgemeine Methoden.*. Springer-Verlag, 2000.

[26] Shneiderman, B.: *Tree Visualization with Tree-Maps: A 2-D Space-Filling Approach*. ACM Transactions on Graphics, 11(1):92–99, 1992.

[27] Shneiderman, B.: *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*. In: *IEEE Visual Languages*, Nr. UMCP-CSD CS-TR-3665, S. 336–343, 1996.

[28] Ware, C.: *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2000.

7.4. Bauhaus

[29] *Abteilung Programmiersprachen und Übersetzerbau, Institut für Softwaretechnologie, Universität Stuttgart*. URL: `http://www.iste.uni-stuttgart.de/ps/`, 17. Okt. 2006.

[30] *AG Softwaretechnik, Universität Bremen*. URL: `http://www.informatik.uni-bremen.de/st/`, 17. Okt. 2006.

[31] *AG Softwaretechnik, Universität Bremen*. URL: `http://www.informatik.uni-bremen.de/st/`, 17. Okt. 2006.

[32] *Axivion Bauhaus Suite*. URL: `http://www.axivion.de/productsaxivionbauhaussuite.html`, 17. Okt. 2006.

[33] *Axivion GmbH*. URL: `http://www.axivion.de`, 17. Okt. 2006.

[34] *Bauhaus*. URL: `http://www.bauhaus-stuttgart.de/bauhaus/`, 17. Okt. 2006.

[35] Bellon, S., D. Simon und G. Vogel: *Emacs mode for CPF format*. URL: `http://www.bauhaus-stuttgart.de/clones/cpf-mode.el`, 17. Okt. 2006.

[36] Ebert, J., K. Kontogiannis und J. Mylopoulus: *Interoperability of Reverse Engineering Tools*. 2001.

[37] Eisenbarth, T., R. Koschke, E. Plödereder, J.-F. Girard und M. Würthner: *Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen*. In: *Workshop Software-Reengineering*, Nr. 7-99, S. 17–26. Bad Honnef, Universität Koblenz-Landau, Fachberichte Informatik, 1999.

[38] Raza, A., G. Vogel und E. Plödereder: *Bauhaus – a Tool Suite for Program Analysis and Reverse Engineering*. URL: `http://www.bauhaus-stuttgart.de/bauhaus/papers/bauhaus.pdf`, 17. Okt. 2006, 2005.

[39] Schuerr, A., S. E. Sim, R. Holt und A. Winter: *Graph eXchange Format*. URL: `http://www.gupro.de/GXL/`, 17. Okt. 2006.