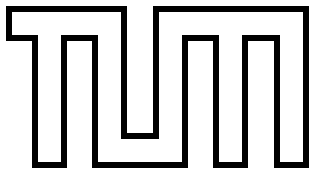


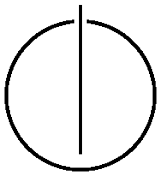
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Seminararbeit zum Proseminar Software-Qualität

Gapped Clone Detection

Marius Daniel Schulz



Abstract

In diesem Paper wird die Problematik inkonsistenter Klone in Software-Quellcode beleuchtet. Es wird diskutiert, welche Inkonsistenzen es gibt und welche Auswirkungen diese zur Folge haben. Weiterhin werden Verfahren zur Erkennung und Entfernung von Klonen dargelegt. Schlussendlich wird anhand einer Analyse von fünf in Java implementierten Open-Source-Systemen aufgezeigt, wie inkonsistente Klone in Real-World-Projekten aussehen können.

Inhaltsverzeichnis

1 Einführung

1.1 Begriffsklärung

1.1.1 Clones

Dem Klonbegriff dieses Papers liegt die Definition von Bettenburg et al. zugrunde, wie er in [3] verwendet wird:

> *A code clone is a part of the source code that is identical, or at least highly similar, to another part (clone) in terms of structure and semantics.*

Da die Frage nach semantischer Äquivalenz i.a. nicht entscheidbar ist, interessieren uns an dieser Stele lediglich diejenigen Klone, die sich in ihrer Programmdarstellung ähneln (engl. *representational clones*).

1.1.2 Gapped Clones

Als *gapped clones* bezeichnet man dabei diejenigen Klone, die sich bis auf geringe Abweichungen ähnlich sind; der Begriff „Klon“ darf daher an dieser Stelle nicht als absolut identische Kopie verstanden werden. Im Abschnitt 3.1 wird im Detail auf mögliche Unterschiede zwischen Code-Fragmenten eingegangen, die hier unter dem recht unspezifischen Begriff „geringe Abweichung“ zusammengefasst sind.

1.2 Warum kopieren wir Code?

Es gibt diverse Gründe, die einen Entwickler dazu bewegen, existierenden Code zu kopieren und somit Copy & Paste-Programmierung zu betreiben.

1.2.1 Limitierungen der Programmiersprache

Bietet die vorliegende Programmiersprache keine Unterstützung für bestimmte Sprachkonstrukte, mit denen sich Code an anderer Stelle wiederverwenden lässt, ist das Kopieren von Code teilweise die einzige Option des Entwicklers.

Ein konkretes Beispiel für eine Sprache, der solch ein Feature fehlte, ist Java. Vor der im September 2004 veröffentlichten Version 5.0 gab es in Java keine Generics. Sollte damals eine bestimmte Funktionalität auf mehreren Datentypen definiert werden, wie z.B. eine Berechnung auf den numerischen Typen `int` und `long`, so musste sie für jeden Datentyp einzeln implementiert werden.

1.2.2 Cross-Cutting Concerns

Als Cross-Cutting Concerns bezeichnet man Belange einer Software, die sich, gemäß ihrem Namen, horizontal durch die Anwendung ziehen. Darunter fallen z.B. die folgenden Aspekte:

- Logging,
- Caching,
- Validierung,
- Autorisierung,
- Lokalisierung,
- Memory Management, uvm.

Weil sich diese Funktionalität teilweise nur schwer an einer Stelle zusammenfassen lässt, wird häufig Copy & Paste eingesetzt, um diese Aspekte an unterschiedlichen Stellen zu implementieren, was verstärkt zu Codeduplizierung führt. Lösungsansätze dafür bietet die Aspektorientierte Programmierung [7].

1.2.3 Anfangs unklare Abstraktionen

Bei der Entwicklung eines Softwaremoduls sind häufig nicht alle sinnvollen Abstraktionen von Anfang an offensichtlich. Das Entwicklerteam kann daher die Phase der Abstraktion nach hinten schieben, bis sich Muster in der Verwendung der Komponente abzeichnen. So lässt sich vorzeitiges *Overengineering* vermeiden, das die Komplexität der Software unnötig erhöht hätte. Der Preis dafür ist die (temporäre) Redundanz, die der duplizierte Code mit sich bringt, mit all ihren Nachteilen, die im folgenden behandelt werden.

1.2.4 Weitere Gründe

Die bisher genannten Gründe waren allesamt technischer Natur, jedoch gibt es darüber hinaus eine Reihe an weiteren Faktoren, die zur Duplizierung von Code beitragen.

Steht das Entwicklungsteam eines Softwareprojektes unter akutem Zeitdruck, kann es als Zwischenlösung lukrativ erscheinen, Code zu duplizieren, anstatt auf saubere Wiederverwendungsmechanismen zu setzen. Findet später kein Refactoring statt, häuft sich *technical debt* in Form von Code-Klonen an. Aus eigener Projekterfahrung kann ich bestätigen, dass solche Provisorien zu ausgeprägter Langlebigkeit tendieren.

Koschke [10] führt an, dass fragwürdige Produktivitätsmetriken von Entwicklern, wie z.B. die Anzahl geschriebener Code-Zeilen, explizit zum Duplizieren von Code verleiten. Schlussendlich ist es ebenfalls denkbar, dass Code Clones aufgrund von Faulheit, Gleichgültigkeit oder mangelndem Bewusstsein der Entwickler stattfindet.

2 Motivation & Relevanz

2.1 Entstehung von Inkonsistenzen

Dupliziert ein Entwickler ein Stück Code, so ist das kopierte Fragment nun an einer weiteren Stelle redundant vorhanden. Damit unterscheidet es sich aber noch nicht vom Ausgangscode, ist also noch eine konsistente Kopie. Eine Reihe an Ursachen kann jedoch für die

Entstehung von Inkonsistenzen sorgen, wie z.B. Code-Vorlagen, Vorsicht des Entwicklers und Versehen.

2.1.1 Inkonsistenz durch Vorlagen

Die Wiederverwendung von existierendem Code per Copy & Paste ist laut Kim et al. [9] die häufigste Ursache fürs Klonen. Dies trifft insbesondere auf Schnittstellen von Funktionsbibliotheken zu, bei deren Verwendung sehr ähnliche Aufrufsmuster entstehen. Nimmt dieser Boilerplate-Code größere Ausmaße an, deutet dies vermutlich auf ein mangelhaftes Design der Schnittstelle hin [11].

Häufig wird jedoch der kopierte Code nicht 1:1 übernommen, sondern abgeändert und an die neuen Bedürfnisse angepasst; es entsteht ein inkonsistenter Klon. Das kopierte Code-Fragment dient demnach als Vorlage, weshalb man von *templating* spricht.

2.1.2 Inkonsistenz aus Vorsicht

Auch die Vorsicht eines Entwicklers kann Ursache für inkonsistente Klone sein. Muss ein Entwickler beispielsweise eine Änderung an einem ihm fremden Legacy-System vornehmen, sind ihm oftmals die Auswirkungen seiner Änderung unbekannt. Häufig besitzen diese Systeme keine Tests, was die Unsicherheit noch befördert. In solchen Fällen kann sich der Entwickler dazu entscheiden, seine Anpassungen in einer Kopie vorzunehmen, damit der existierende Code unangetastet bleibt und Breaking Changes vermieden werden.

2.1.3 Inkonsistenz aus Versehen

Sowohl bei der Wiederverwendung durch Vorlagen als auch bei der Code-Duplizierung und -Anpassung aus Vorsicht hat der Entwickler inkonsistente Klone bewusst in Kauf genommen. Jedoch kann es leicht passieren, ungewollte Inkonsistenzen herbeizuführen, indem bei einer Änderung an einem Klon nicht alle Instanzen seiner sogenannten *Klongruppe* angepasst werden. Der Entwickler hat so mangels Kenntnis der anderen Ausprägungen dieses Klons unbewusst inkonsistenten Code erzeugt.

2.2 Wartungsaufwand und Fehlverhalten

Klone im Allgemeinen führen zu erhöhtem Wartungsaufwand [10], da jede Ausprägung eines Klons gefunden und abgeändert werden muss, um Inkonsistenzen zu vermeiden. Da in der Regel jedoch nicht dokumentiert ist, wo Code geklont worden ist [11], müssen Klone zur Wartungszeit gefunden werden. Schon bei mittelmäßig großen Projekten ist das manuell nicht mehr sinnvoll machbar, sodass automatisierte Unterstützung durch Tooling benötigt wird [11].

Wird die Anpassung bei einem oder mehreren Klonen der Klongruppe vergessen oder übersehen, kann die enstandene Inkonsistenz leicht zu unerwartetem Verhalten des Pro-

gramms führen: Ein Teil der Logik des Systems ist nun an verschiedenen Stellen unterschiedlich implementiert – eine Steilvorlage für Bugs und damit zentraler Bestandteil des Problems.

2.2.1 Empirische Studien

Dass das bei weitem kein seltener Fall ist, zeigen Juergens et al. eindrucksvoll in [8]. In ihrer Studie haben die Autoren fünf Software-Systeme untersucht, die in C#, Java und COBOL implementiert wurden. Im Abschnitt „Results“ schlüsseln sie auf, wie viele Klongruppen die Systeme enthielten, welche davon inkonsistent waren und welche wiederum davon unbeabsichtigt waren.

Die beeindruckendste Ziffer schließlich ist der Prozentsatz der unbeabsichtigt inkonsistenten Klone (orig. *unintentionally inconsistent clones*), die zu einem Fehlverhalten (einem *Fault*) des Programms führten: Laut Studie beläuft sich dieser dort als $\frac{|F|}{|UIC|}$ bezeichnete Wert auf 50%. Intuitiv formuliert heißt das, dass duplizierter Code, der unabsichtlich von seinen anderen Ausprägungen abweicht, in jedem zweiten Fall Ursache für ungewolltes Fehlverhalten ist. Nach ihren Untersuchungen schlussfolgern die Autoren, dass der Anteil an Fehlern in inkonsisten Klonen höher ist als im Schnitt und bekräftigen somit die Relevanz der Klonfindung.

Andere Autoren auf dem Gebiet bestätigen dieses Ergebnis. Monden et al. [12] berichten von einer höheren Fehlerdichte in Software-Modulen mit großen Klonen. Chou et al. [4] stellten fest, dass Code mit einem Fehler überdurchschnittlich häufig weitere Fehler aufweist, wenn er Klone enthält.

Zusammenfassend lässt sich festhalten, dass Klone – allen voran unbeabsichtigt inkonsistente Klone – einen negativen Einfluss auf die Qualität einer Software haben. Sie machen die Wartung einer Anwendung teurer und erhöhen die Wahrscheinlichkeit, dass sich dabei Fehler einschleichen. Im Sinne der Qualitätssicherung ist es daher im Interesse der Entwickler, Klone im Quellcode automatisiert zu finden.

3 Erkennung von Klonen

3.1 Klon-Typen

Bevor wir uns verschiedenen Techniken zur Klonerkennung widmen, müssen wir uns bewusst machen, was es für Arten von Klonen gibt und worin sie sich unterscheiden. In der Literatur [10] werden Klone in Software-Quellcode wie folgt kategorisiert:

Typ 1: Exakte Kopie bis auf Whitespace, Formatierung und Kommentare.

Typ 2: Syntaktisch identische Kopie, bei der Literalwerte abgeändert oder Bezeichner von Variablen, Typen und Funktionen umbenannt wurden.

Typ 3: Kopie mit eingefügten, veränderten oder gelöschten Statements.

Als *gapped clones* bezeichnet man Klone vom Typ 3. Das „gapped“ im Namen steht dabei für die Abweichung der Kopien voneinander, also die symbolische „Lücke“.

3.2 Precision & Recall

Bei der Betrachtung verschiedener Techniken zur Klonerkennung ist von Interesse, welche Genauigkeit und Trefferquote sie aufweisen. In der englischsprachigen Literatur werden diese Aspekte als *precision* und *recall* bezeichnet. Während *precision* den Anteil relevanter Ergebnisse an allen Suchergebnissen angibt, misst *recall* die Vollständigkeit der gefundenen Resultate. Beim Nachschlagen in der Literatur begegnet man häufig diesen beiden Begriffen, weshalb sie hier der Vollständigkeit halber erwähnt seien.

3.3 Techniken zur Klonerkennung

3.3.1 Textbasiert

Beim Textvergleich werden zwei Klonkandidaten zeilenweise miteinander verglichen. Damit minimale Änderungen an der Darstellung eines Code-Fragments nicht sofort zum Scheitern des String-Vergleichs führen, werden beim Vergleich häufig Kommentare und Whitespace ignoriert.

Da String-Vergleiche generell vergleichweise teuer sind, kann zur Steigerung der Performance eine Hash-Funktion eingesetzt werden. Diese berechnet für jede Zeile einen Hash und unterteilt somit die Menge aller Zeilen in verschiedene Partitionen mit dem gleichen Hash-Wert. Der anschließende String-Vergleich wird lediglich auf Zeilen innerhalb der gleichen Partition angewendet [10].

3.3.2 Token-basiert

Beim Token-basierten Ansatz sind nicht mehr die einzelnen Zeichen zweier Code-Fragmente die kleinste Einheit, die zum Vergleich herangezogen wird, sondern deren Tokens [1]. Genau wie beim textbasierten Ansatz findet hierbei i.d.R. eine Normalisierung statt, sodass nicht die konkreten Werte eines Tokens betrachtet werden, sondern nur dessen Identität. Damit wird die Token-basierte Erkennung robust gegen Modifizierungen von Whitespace oder Kommentaren, Umbenennung von Identifiern und Änderung von Literalwerten.

Zum effizienten Finden gleicher Token-Sequenzen eignet sich als Datenstruktur ein *suffix tree*, der sich in linearer Laufzeit konstruieren lässt [11]. Klonkandidaten mit bestimmter Mindestlänge lassen sich von der Wurzel ausgehend leicht ablesen, da die Pfade zu den Blättern mit den gemeinsamen Präfixen beschriftet sind; diese Präfixe repräsentieren den geklonten Teil der Code-Fragmente und machen so den Klonkandidaten aus.

3.3.3 Syntaxbaum-basiert

Wie der Name vermuten lässt, wird beim Syntaxbaum-basierten Ansatz der abstrakte Syntaxbaum (engl. AST für *abstract syntax tree*) eines Programms erstellt. Die Robustheit der Token-basierten Erkennung gegen (strukturerhaltende) Änderungen ist auch hier gegeben. Der eigentliche Vergleich findet hier auf Teilbäumen des Syntaxbaums statt, die im Sinne der Performancesteigerung gehasht werden können.

Anders als beim Text- oder Token-basierten Ansatz ist für den Aufbau des Syntaxbaums ein tieferes Programmverständnis erforderlich: Es wird ein Parser benötigt, der die Grammatik der verwendeten Programmiersprache versteht. Damit der Parser einen Syntaxbaum erstellen kann, muss der Code in kompilierten Sprachen syntaktisch und semantisch korrekt sein; fehlerhafter Code, der nicht kompiliert, kann also nicht mithilfe eines Syntaxbaums auf Klone untersucht werden.

3.3.4 Abhängigkeitsgraph-basiert

Das letzte Verfahren, das in diesem Paper betrachtet werden soll, baut auf dem sogenannten Abhängigkeitsgraph (*program dependence graph*, oder kurz PDG) eines Programms auf. Es macht sich zunutze, dass Zeilen bzw. Statements eines Code-Fragmentes i.d.R. nicht beliebig vertauscht werden können, wenn sie gleichzeitig semantisch korrekt und sinnerhaltend bleiben sollen.

Imperative Programme bestehen aus einer Reihe von Statements, von denen manche Abhängigkeiten untereinander aufweisen. So kann in C-ähnlichen Sprachen eine Variable erst nach ihrer Deklaration verwendet werden, aber nicht vorher. Auch bestehen logische Abhängigkeiten zwischen Statements: Wird ein numerischer Parameter einer Methode erst mit einem Faktor multipliziert und anschließend zu einer Konstanten dazuaddiert, liefern die Operationen in umgekehrter Ausführungsreihenfolge i.a. nicht das gleiche Ergebnis. Die zwei entsprechenden Statements können demnach nicht vertauscht werden, ohne die Programmlogik zu ändern.

Ein *program dependence graph* ermittelt nun die Abhängigkeiten zwischen den Statements eines Code-Fragmentes und repräsentiert diese als Graph. Der Vergleich zweier Code-Einheiten wird dann als Suche nach isomorphen Teilgraphen durchgeführt. Dieses Problem ist jedoch NP-schwer, weshalb in der Praxis approximative Verfahren eingesetzt werden [10].

3.3.5 Weitere Techniken

Die Liste der genannten Techniken zur Klonerkennung ist natürlich nicht vollständig; so wurden z.B. Metriken-basierte Ansätze nicht besprochen. Es existiert eine Vielzahl an Algorithmen und Verfahren für dieses Gebiet, aber auch für verwandte Gebiete wie die Plagiatserkennung. Diese würden jedoch den Rahmen dieser kurzen Arbeit sprengen und sind daher in der Literatur nachzulesen.

4 Beseitigung von Klonen

Eine Strategie zur Entfernung neuer Klone kann es sein, diese gar nicht erst entstehen zu lassen. Balazinska et al. [2] führen dafür Entwurfsmuster als Ansatz an, der die Anzahl an Klonen durch eine bessere Architektur verringert. Dies ist jedoch nur schwer automatisiert machbar und muss daher i.d.R. manuell erledigt werden [10].

4.1 Auslagerung von geteiltem Code

Sind die Klone bereits vorhanden, so können diese häufig durch Refactoring-Verfahren wie *Extract Method* oder *Extract Class* beseitigt werden, sofern die verschiedenen Ausprägungen nicht zu sehr voneinander abweichen [6]. Zum Behandeln von Typ-3-Klonen nennt Koschke beispielhaft Makros und Präprozessor-Anweisungen als Lösungsansatz [10].

Aus meiner eigenen Erfahrung möchte ich an dieser Stelle darauf hinweisen, dass Direktiven für einen Präprozessor – insbesondere Verfahren wie die bedingte Kompilierung – mit Vorsicht zu genießen sind. Auf der einen Seite sind sie schwer zu testen, da der Code in allen möglichen Varianten kompiliert werden muss. Auf der anderen Seite steigern sie die Komplexität eines Moduls, sodass dessen Verständlichkeit leidet.

4.2 Automatische Code-Generierung

Abschließend ist es möglich, sich codegenerierende Verfahren zunutze zu machen und den redundanten Code automatisiert erzeugen zu lassen. Da die Generierung nicht manuell stattfindet, ist sie nicht anfällig für Flüchtigkeits- oder Tippfehler. Java setzt diese Methode beispielsweise ein, um den Quellcode für verschiedene Buffer-Klassen wie `ByteBuffer` und `CharBuffer` automatisiert zu erstellen [11]. Dafür kommt XVCL zum Einsatz, die *XML-Based Variant Configuration Language* [13]. Diese manipuliert konfigurierbare Programmkomponenten gemäß einer XML-Beschreibung und macht so die manuelle Anpassung unnötig.

4.3 Kosten-Nutzen-Verhältnis

Damit die Ressourcen, die für die Entfernung von Klonen aufgewendet werden, gerechtfertigt werden können, müssen Kosten und Nutzen der Entfernung in einem günstigen Verhältnis stehen. Es muss vorab situationsabhängig entschieden werden, ob im betroffenen Modul zum gegebenen Zeitpunkt eine Entfernung überhaupt gewünscht ist. Wenn bereits geplant ist, dass sich in naher Zukunft beide Kopien unabhängig voneinander in verschiedene Richtungen weiterentwickeln, macht eine Zusammenführung wenig Sinn. Im Falle solch einer Weiche spricht man daher treffend von *forking* [10].

5 Analysierte Projekte

Mithilfe von ConQat [5], dem *Continuous Quality Assessment Toolkit*, wurden für diese Arbeit die folgenden fünf quelloffenen Java-Projekte auf inkonsistente Klone untersucht:

- Art of Illusion (AoI)
- ArgoUml
- FreeCol
- FreeMind
- JUnit

Für die Analyse wurde der ConQat-Block `JavaGappedCloneAnalysis` gewählt. Zu Beginn wurden für die Parameter *gap ratio*, *clone minlength* und *max errors* die Werte 0.25, 10 und 5 verwendet. Bei der Suche nach Klonen hat sich schnell herausgestellt, dass die Qualität der gefundenen Resultate maßgeblich von den gewählten Parametern abhängt. Während eine vorgeschriebene Mindestlänge von 10 bei AoI gut funktioniert und längere Klone freigelegt hat, ist dieser Wert deutlich zu hoch für JUnit gewesen: Die meisten Methoden in JUnit sind kurz gehalten und weisen daher nur deutlich kürzere Klone auf. Für ein Projekt, das sich der Qualitätssicherung durch Tests verschrieben hat, ist diese Erkenntnis nicht weiter verwunderlich.

Der mit weit über 100 Zeilen längste gefundene Klon war in AoI enthalten. Bei einer experimentellen Erhöhung des Wertes *max errors* auf 25 fand ihn ConQat mit 24 Gaps. Im Anhang sind dessen Beginn sowie weitere beispielhafte Klone aus den analysierten Projekten zu finden.

Literatur

[1] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.

[2] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 98–107. IEEE, 2000.

[3] Nicolas Bettenburg, Weiyi Shang, Walid M Ibrahim, Bram Adams, Ying Zou, and Ahmed E Hassan. An empirical study on inconsistent changes to code clones at the release level. *Science of Computer Programming*, 77(6):760–776, 2012.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. *An Empirical Study of Operating Systems Errors*, volume 35. ACM, 2001.

[5] Florian Deissenboeck, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt Mas y Parareda, and Markus Pizka. Tool support for continuous quality control. *Software, IEEE*, 25(5):60–67, 2008.

[6] Richard Fanta and Václav Rajlich. Removing clones from the code. *Journal of Software Maintenance*, 11(4):223–243, 1999.

[7] Robert E. Filman, Siobhan Clarke, and Tzilla Elrad. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[8] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

[9] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 83–92. IEEE, 2004.

[10] Rainer Koschke. *Survey of Research on Software Clones*. 2007.

[11] Rainer Koschke. Similarity in software artifacts and its relation to code generation. April 2013. Code Generation 2013.

[12] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and K-i Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 87–94. IEEE, 2002.

[13] National University of Singapore. XML-based variant configuration language. `http://xvcl.comp.nus.edu.sg/cms/`. Abgerufen: 12. Juni 2014.

6 Anhang: Beispielhafte Klone

clonedetection-aoi.cqr | Clone Compare View | Clone Compare View

AOI/view/SoftwareCanvasDrawer.java

```
516   private Vec2 [] clipTriangle(Vec3 v1, Vec3 v2, Vec3 v3, double z1, double z2, d
517   {
518     double clip = cam.getClipDistance();
519     Mat4 toScreen = cam.getObjectToScreen();
520     boolean c1 = z1 < clip, c2 = z2 < clip, c3 = z3 < clip;
521     Vec3 u1, u2, u3, u4;
522     int clipCount = 0;
523
524     if (c1) clipCount++;
525     if (c2) clipCount++;
526     if (c3) clipCount++;
527     if (clipCount == 2)
528       {
529         // Two vertices need to be clipped.
530
531         if (!c1)
532           {
533             u1 = v1;
534             newz[0] = z1;
535             double f2 = (z1-clip)/(z1-z2), f1 = 1.0-f2;
536             u2 = new Vec3(f1*v1.x+f2*v2.x, f1*v1.y+f2*v2.y, f1*v1.z+f2*v2.z);
537             newz[1] = f1*z1 + f2*z2;
538             f2 = (z1-clip)/(z1-z3);
539             f1 = 1.0-f2;
540             u3 = new Vec3(f1*v1.x+f2*v3.x, f1*v1.y+f2*v3.y, f1*v1.z+f2*v3.z);
541             newz[2] = f1*z1 + f2*z3;
542           }
543         else if (!c2)
544           {
545             u2 = v2;
546             newz[1] = z2;
547             double f2 = (z2-clip)/(z2-z3), f1 = 1.0-f2;
548             u3 = new Vec3(f1*v2.x+f2*v3.x, f1*v2.y+f2*v3.y, f1*v2.z+f2*v3.z);
549             newz[2] = f1*z2 + f2*z3;
550             f2 = (z2-clip)/(z2-z1);
551             f1 = 1.0-f2;
552             u1 = new Vec3(f1*v2.x+f2*v1.x, f1*v2.y+f2*v1.y, f1*v2.z+f2*v1.z);
553             newz[0] = f1*z2 + f2*z1;
554           }
555         else
556           {
557             u3 = v3;
558             newz[2] = z3;
559             double f2 = (z3-clip)/(z3-z1), f1 = 1.0-f2;
560             u1 = new Vec3(f1*v3.x+f2*v1.x, f1*v3.y+f2*v1.y, f1*v3.z+f2*v1.z);
561             newz[0] = f1*z3 + f2*z1;
562             f2 = (z3-clip)/(z3-z2);
563             f1 = 1.0-f2;
564             u2 = new Vec3(f1*v3.x+f2*v2.x, f1*v3.y+f2*v2.y, f1*v3.z+f2*v2.z);
565             newz[1] = f1*z3 + f2*z2;
566           }
567         return new Vec2 [] {toScreen.timesXY(u1), toScreen.timesXY(u2), toScreen.
568       }
569
570     // Only one vertex needs to be clipped, resulting in a quad.
571
```

AOI/view/SoftwareCanvasDrawer.java

```
772
773   private Vec2 [] clipSmoothTriangle(Vec3 v1, Vec3 v2, Vec3 v3, double z1, doub
774   {
775     double clip = cam.getClipDistance();
776     Mat4 toScreen = cam.getObjectToScreen();
777     boolean c1 = z1 < clip, c2 = z2 < clip, c3 = z3 < clip;
778     Vec3 u1, u2, u3, u4;
779     int clipCount = 0;
780
781     if (c1) clipCount++;
782     if (c2) clipCount++;
783     if (c3) clipCount++;
784     if (clipCount == 2)
785       {
786         // Two vertices need to be clipped.
787
788         if (!c1)
789           {
790             u1 = v1;
791             newz[0] = z1;
792             newc1.copy(col1);
793             double f2 = (z1-clip)/(z1-z2), f1 = 1.0-f2;
794             u2 = new Vec3(f1*v1.x+f2*v2.x, f1*v1.y+f2*v2.y, f1*v1.z+f2*v2.z);
795             newc2.setRGB(f1*col1.getRed()+f2*col2.getRed(), f1*col1.getGreen()+
796             newz[1] = f1*z1 + f2*z2;
797             f2 = (z1-clip)/(z1-z3);
798             f1 = 1.0-f2;
799             u3 = new Vec3(f1*v1.x+f2*v3.x, f1*v1.y+f2*v3.y, f1*v1.z+f2*v3.z);
800             newc3.setRGB(f1*col1.getRed()+f2*col3.getRed(), f1*col1.getGreen()+
801             newz[2] = f1*z1 + f2*z3;
802           }
803         else if (!c2)
804           {
805             u2 = v2;
806             newz[1] = z2;
807             newc2.copy(col2);
808             double f2 = (z2-clip)/(z2-z3), f1 = 1.0-f2;
809             u3 = new Vec3(f1*v2.x+f2*v3.x, f1*v2.y+f2*v3.y, f1*v2.z+f2*v3.z);
810             newc3.setRGB(f1*col2.getRed()+f2*col3.getRed(), f1*col2.getGreen()+
811             newz[2] = f1*z2 + f2*z3;
812             f2 = (z2-clip)/(z2-z1);
813             f1 = 1.0-f2;
814             u1 = new Vec3(f1*v2.x+f2*v1.x, f1*v2.y+f2*v1.y, f1*v2.z+f2*v1.z);
815             newc1.setRGB(f1*col2.getRed()+f2*col3.getRed(), f1*col2.getGreen()+
816             newz[0] = f1*z2 + f2*z1;
817           }
818         else
819           {
820             u3 = v3;
821             newz[2] = z3;
822             newc3.copy(col3);
823             double f2 = (z3-clip)/(z3-z1), f1 = 1.0-f2;
824             u1 = new Vec3(f1*v3.x+f2*v1.x, f1*v3.y+f2*v1.y, f1*v3.z+f2*v1.z);
825             newc1.setRGB(f1*col3.getRed()+f2*col1.getRed(), f1*col3.getGreen()+
826             newz[0] = f1*z3 + f2*z1;
827             f2 = (z3-clip)/(z3-z2);
```

Abbildung 1: Beginn eines langen Klons in AoI

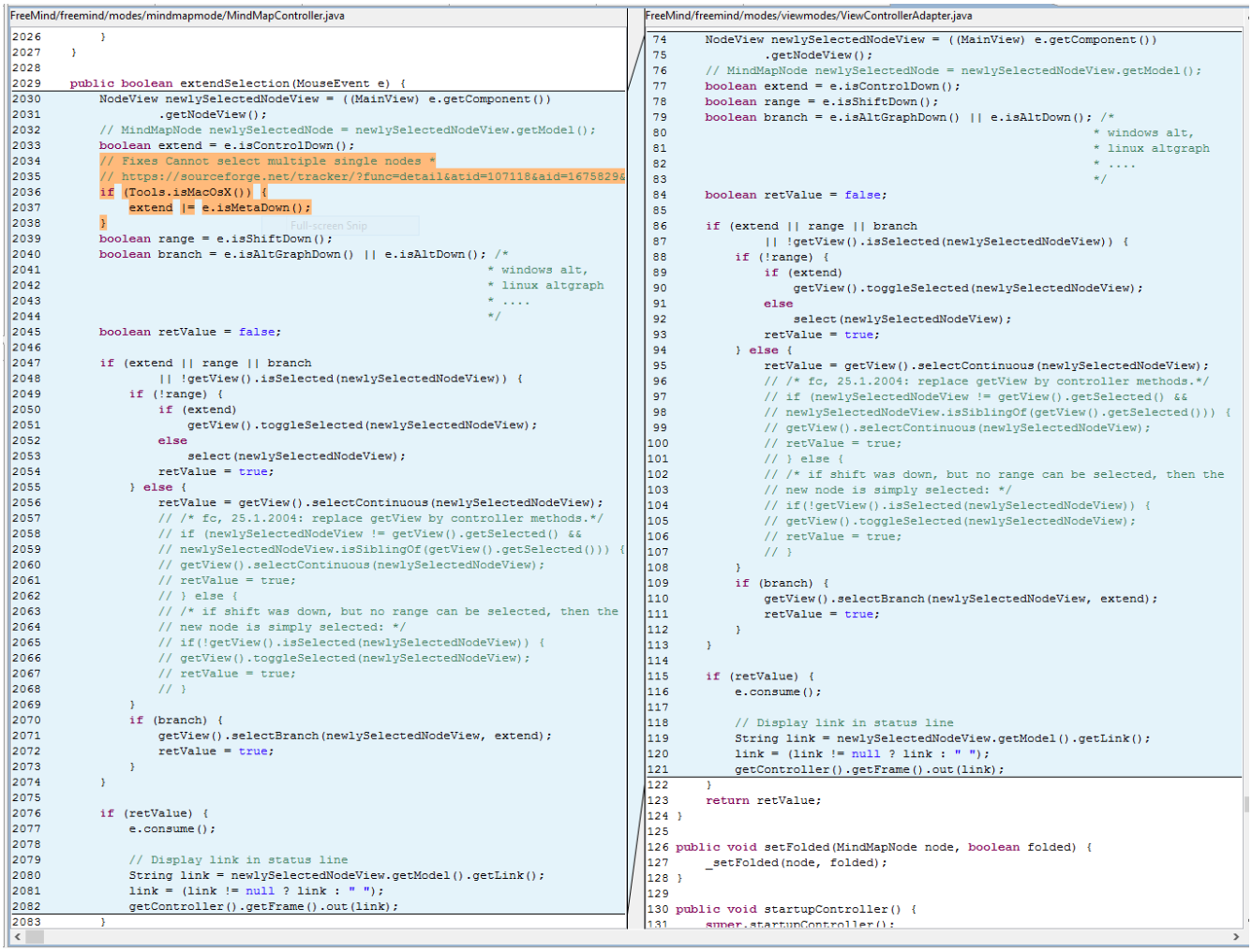


Abbildung 2: Möglicherweise unabsichtlich inkonsistenter Bugfix in FreeMind (#1)

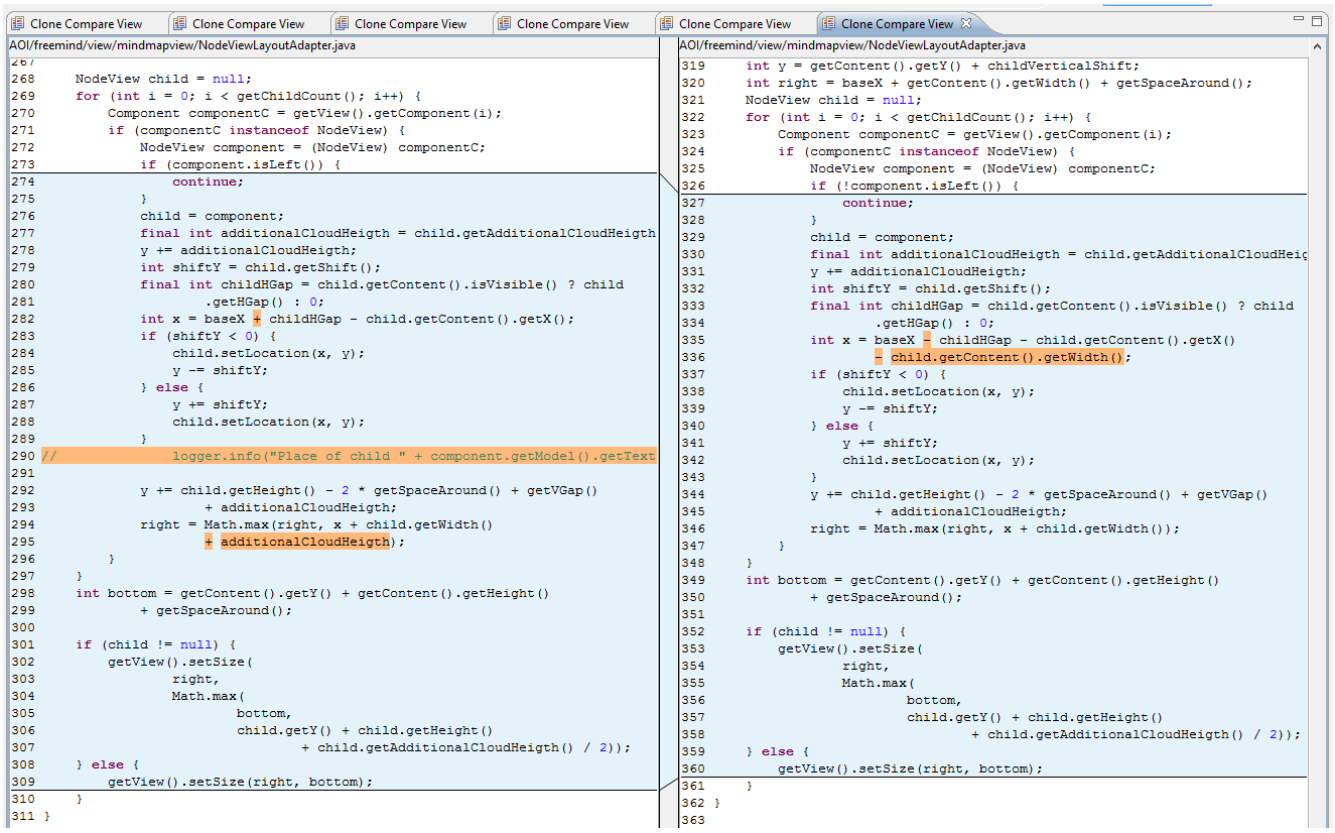


Abbildung 3: Möglicherweise unabsichtlich inkonsistenter Bugfix in FreeMind (#2)

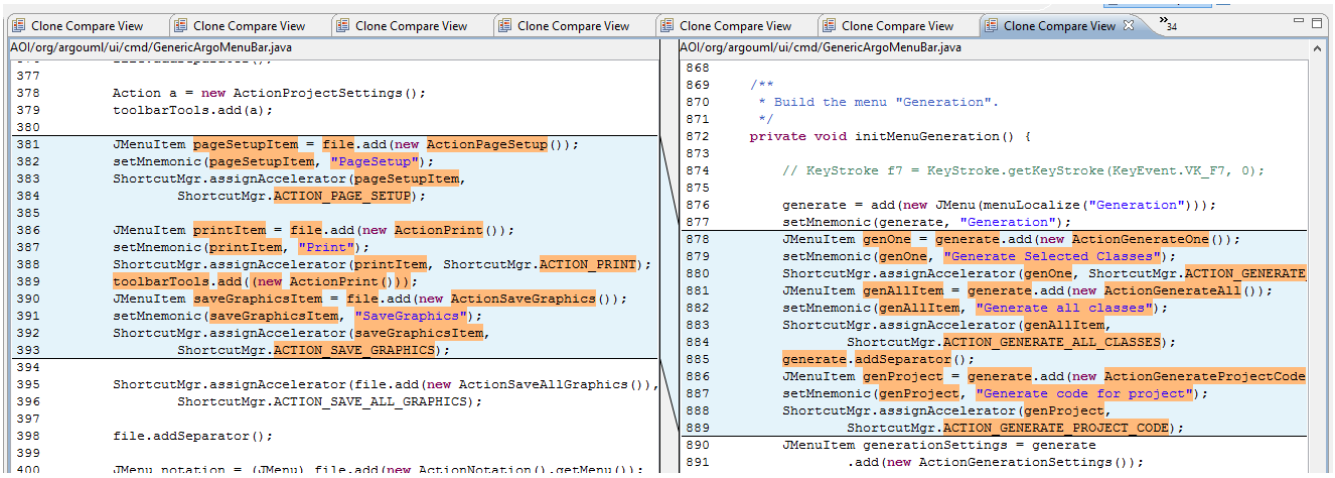


Abbildung 4: Ein *false positive* in ArgoUml

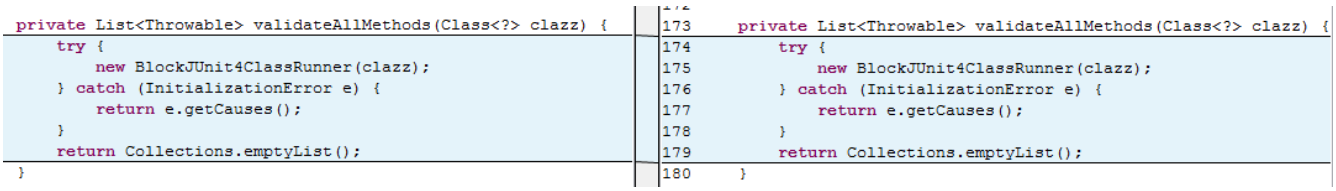


Abbildung 5: Ein uninteressanter Klon ohne Gaps in JUnit