修士学位論文

題目

Java における hashCode メソッドの整合性検査手法の精度に関する調査および改善案の提案

指導教員

楠本 真二 教授

報告者

藤田 悠矢

平成 27 年 2 月 6 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

平成 26 年度 修士学位論文

Java における hashCode メソッドの整合性検査手法の精度に関する調査および改善案の提案

藤田 悠矢

内容梗概

Java において，コレクションに格納されるオブジェクトは equals メソッドと hashCode メソッドをオーバーライドしている必要があり，これらのメソッドには満たすべき性質 (反射性，対称性，推移性など) が存在する．この性質を満たさないオブジェクトをコレクションに格納して使用した場合，正しい振る舞いをしなくなってしまい，さらにはそれによって引き起こされる欠陥を発見することが難しくなる．

既存研究では equals メソッドのみを対象として，性質を満たすかどうかを検査する手法が提案されていたため，当研究室では hashCode メソッドを対象としたモデル検査器による整合性検査手法を提案した．その手法では，モデル化の困難な性質に対してはサブセット規則を提案して検査を行った．しかしこの規則は厳しすぎるため，もとの性質に違反していない hashCode メソッドを違反する，誤検出が生じてしまう．

そのため，本論文では多数のプロジェクトに対してサブセット規則による検査を行い，サブセット規則の精度に関する調査を行った．また，調査によって発見した誤検出に対して，改善方法の考察を行った．

主な用語

Java，hashCode, Empirical Study, Consistency Checking

目 次

図 目 次

表 目 次

1　はじめに

Java において，オブジェクト同士の等価判定を行うクラスでは equals メソッドをオーバーライドする．また，equals メソッドをオーバーライドする場合には hashCode メソッドもオーバーライドする必要がある [1]．Java の Object クラスには，Oracle の API 仕様によって，これらのメソッドに対して満たさなければならない性質が規定されている [2]．例えば equals メソッドでは反射性，対称性，推移性を満たしている必要がある．これらの性質に違反したクラスの実装は欠陥を誘発するものであり，そのクラスのオブジェクトをコレクションに格納して使用すると，正しい振る舞いをせず，これにより誘発される欠陥を見つけることが難しくなる [1][3][4][5].

このような性質の違反をチェックする手法としては Rupakheti らによる検査手法が存在する [6] [7] [8] [9]．この手法では Java の equals メソッドを対象として，反射性，対称性，推移性などの性質に違反しているかどうかの検査を軽量的に行っている．しかし，この研究では equals メソッドのみの検査を行っており，equals メソッドがオーバーライドされるクラスにおいて，同様にオーバーライドされなければならない hashCode メソッドの検査は行っていない．

当研究グループでは Java を対象として hashCode メソッドが満たすべき性質に違反しているかどうかを検査する手法を提案し [10]，その実装を行った [11]．hashCode メソッドの満たすべき性質は equals メソッドに依存しているため，equals メソッドの検査もあわせて行っている．また，ハッシュコードの計算中で使用されることが多いビット演算に対応するために SMT ソルバの Z3 を用いて検査を行っている．equals メソッドと hashCode メソッド中で行われる処理のパターンは異なるため，変換する処理のパターンの提案と Java コードの SMT-LIB への変換方法の提案を新たに行っている．しかし，hashCode の満たすべき性質のひとつである一貫性に関しては SMT-LIB への変換が困難であるため，条件をより厳しくしたサブセット規則を使用して検査を行っている．しかし，サブセット規則は一貫性よりも厳しいために，一貫性は満たしているのにサブセット規則を満たさないという，誤検出が存在する．誤検出が多い場合，ユーザはサブセット規則による検査の結果違反として出力された，一貫性を満たしている hashCode メソッドに対してもチェックを行う必要があるため，無駄なコストが生じてしまう．[11] では実装したツールを既存のプロジェクトに適用することで検出される違反についての評価を行っているが，対象としたプロジェクト数が少なく，また規模も小さいために十分な調査が行えているとはいえない．大規模な調査を行おうにも eclipse のプラグインであるため eclipse 上で開発しているプロジェクトしか対象として選べない．

そこで，本論文では hashCode メソッドがサブセット規則を満たしているかどうかを検査するツールを開発し，Apache Software Foundation [12] で公開されている 130 のプロジェクトを対象として検査を行うことで，サブセット規則の精度及び誤検出に関する調査を行う．また，発見した誤検出に対してはその改善方法の考察，提案を行う．一部の誤検出に関しては提案した改善方法をツールに実装している．

以降，2 章では研究の背景となる諸技術や当研究室の先行研究について述べる．3 章では研究動機

について説明する．4 章では本調査のために実装したツールに関して説明を行い，5 章では今回行った調査とその結果について述べる．6 章では調査結果を踏まえた改善案等の考察を述べる．最後に 7 章で本報告のまとめを述べる．

2 研究背景

本章では，本研究に関連する概念，ツール，研究について簡単に述べる．

2.1 Java の Object クラス

Java の Object クラスはクラス階層のルートであり，すべてのクラスはスーパークラスとして Object クラスを持っている．配列を含むすべてのオブジェクトは Object クラスのメソッドを実装する．よって equals メソッドや hashCode メソッドをオーバーライドしていないクラスのオブジェクトに対して，equals メソッドや hashCode メソッドを呼び出すと Object クラスのメソッドが呼び出される．

2.1.1 equals メソッド

Object クラスの equals メソッドは public boolean equals(Object obj) と定義されており，呼び出されたオブジェクトと引数で与えられるオブジェクトが等価であるかどうかを示すメソッドである．equals メソッドは null 以外のオブジェクトにおいて以下のような規則 (抜粋) を満たす必要がある．

- 反射性 (reflexive):null 以外の参照値 x について，x.equals(x) は true を返す
- 対称性 (symmetric):null 以外の参照値 x と y について，x.equals(y) は、y.equals(x) が true を返す場合だけ true を返す
- 推移性 (transitive):null 以外の参照値 x，y，z について、x.equals(y) が true を返し，かつ y.equals(z) が true を返す場合に，x.equals(z) は true を返す
- null でない任意の参照値 x について，x.equals(null) は false を返す

Object クラスの equals メソッドはもっとも比較しやすいオブジェクトの同値関係を実装している．null 以外の参照値 x と y について 2 つのオブジェクト x と y が同じオブジェクトを参照する (x == y が true) 場合にだけ true を返す実装となっている．この実装は満たさなければならない規則をすべて満たしている．また，通常 equals メソッドをオーバーライドする場合は，hashCode メソッドを常にオーバーライドして，等価なオブジェクトは等価なハッシュコードを保持する必要があるという hashCode メソッドの規則に従う必要がある．

2.1.2 hashCode メソッド

Object クラスの hashCode メソッドは public int hashCode() と定義されており，オブジェクトのハッシュコードを返すメソッドである．このメソッドは java.util.Hashtable によって提供されるようなハッシュテーブルで使用するために用意されている．hashCode メソッドが満たすべき規則 (抜粋) は以下のように規定されている．ここで情報とは，equals メソッド中で呼び出されたメソッドの戻り値や使用されているフィールドの値などを表す．

```
public class Sample{
 private int val;
 private String str;

 public boolean equals(Object obj){
  if (obj == null)
   return false;
  if (this == obj)
   return true;
  if (!(obj instanceof Sample))
   return false;
  Sample that = (Sample) obj;
  if (this.str == null){
   return that.str == null;
  }
  return this.val == that.val && this.str.equals(that.str)
 }

 public int hashCode(){
  return val + (this.str == null ? 0 : this.str.hashCode());
 }
}
```

図 1: equals メソッドと hashCode メソッドの実装例

- Java アプリケーションの実行中に同じオブジェクト上で複数回呼び出される場合は必ず，このオブジェクトに対する equals による比較で使われた情報が変更されていなければ，hashCode メソッドは同じ整数を一貫して返さなければならない．
- equals(Object) メソッドで 2 つのオブジェクトが等価とされた場合，どちらのオブジェクトで hashCode メソッドを呼び出しても結果は同じ整数値にならなければならない

Object クラスの hashCode メソッドは異なるオブジェクトについては異なる整数値を返す．これは，オブジェクトの内部アドレスを整数値に変換する形で実装されている．この実装は hashCode メソッドが満たすべき規則をすべて満たしているが，これらの規則は equals メソッドの実装に依存することに留意しなければならない．

また，equals メソッドと hashCode メソッドの実装例を図 1 に示す．このクラスは int 型のフィールド val と String 型のフィールド str を持っている．equals メソッドでは，引数で与えられたオブジェクトが自分と等しいかのチェックを行った後，Sample クラスのインスタンスであるかのチェックを行っている．次に自分の str が null の場合は，相手の str も null かのチェックを行い，最後に val の値が等しいかのチェックと str が指している文字列が等しいかのチェックを行っている．hashCode メソッドでは val の値と str のハッシュコードを足し合わせている．この実装は equals メソッドと hashCode メソッドの満たすべき規則に従った実装である．

等価性を満たさない hashCode メソッドの実装の具体例を図 2 に示す．これは Apache の PDFBox に対して報告された欠陥である．equals メソッドでは java.util.Arrays の equlas メソッドにより，byte[]

```
public class  COSString extends COSBase{

 public byte[] getBytes(){
  …
 }

 public boolean equals(Object obj){
  return (obj instanceof COSString)
    && java.util.Arrays.equals(((COSString)obj).getBytes(),getBytes())
 }

 public int hashCode(){
  return getBytes().hashCode();
 }
}
```

図 2: hashCode メソッドの等価性を満たさない例

型の配列の要素数と各要素の値が等しいかチェックしているのに対して，hashCode メソッドでは Java.util.Arrays の hashCode メソッドではなく配列 (byte[]) の hashCode メソッドを呼び出してしまっている．配列の hashCode メソッドではオブジェクトのメモリアドレスを整数に変換した値を返すため，equals メソッドで等しいと判断されてもインスタンスが異なれば hashCode の値が異なるオブジェクトが存在してしまう．このように異なるインスタンスで，equals メソッドで等価と判定された 2 つのオブジェクトを HashSet に入れようとした場合，ハッシュコードの値が異なるのでどちらも HashSet に格納されてしまう．しかし，この状態では値の重複が許されない Set に等価なオブジェクトが 2 つ入ってしまい正しい振る舞いをしなくなってしまう．

一貫性を満たさない hashCode メソッドの実装の具体例も図 3 に示す．これは Apache の xerces における実装を簡略化したものである．equals メソッドではフィールドの uri と localpart が等しいかどうかをチェックしているのに対して，hashCode メソッドでは uri と localport，rawname を使用して値を計算している．uri が null のとき，hashCode メソッドでは rawname を利用して値を計算しているが，この情報は equals メソッドでは使用されていない．そのため，rawname が変わった場合，equals メソッドで使用されている情報が変わっていないのに hashCode メソッドの返す値が変わる可能性があるため，一貫性に違反してしまっている．

2.2 JDT

Java Development Tool(JDT)[13] とは，Eclipse Foundation によってオープンソースソフトウェアとして提供されている．通常 Eclipse のプラグインとして提供されている．JDT は Java のソースコードを AST(Abstruact Syntax Tree) という中間表現を用いて保持しており，この AST をたどる

```
private static final class XQName {
    public String prefix;
    public String localpart;
    public String rawname;
    public String uri;
    public XQName(String prefix, String localpart, String rawname, String uri) {
        setValues(prefix, localpart, rawname, uri);
    }
    …
    public boolean equals(Object object) {
        if (object instanceof QName) {
            QName qname = (QName)object;
            return uri == qname.uri && localpart == qname.localpart;
        }
        return false;
    }
    public int hashCode() {
        if (uri != null) return uri.hashCode() +
                                ((localpart != null) ? localpart.hashCode() : 0);
        return (rawname != null) ? rawname.hashCode() : 0;
    }
}
```

図 3: hashCode メソッドの一貫性を満たさない例

ことにより必要な情報を得ることができる．AST の各ノードは Java の要素を表しており，ASTNode という型を継承したクラスに格納されている．

今回，サブセット規則を判定するツールの実装にはこの JDT を用いて行っている．

2.3 関連研究

本節では本研究に関連する既存研究について説明する．

まず，Object クラスのメソッドの設計や実装に関連する既存研究について説明する．equals メソッドと hashCode メソッドを自動的に生成する手法がいくつか提案されている．Rayside らは equals メソッドや hashCode メソッドの計算に必要なクラスやフィールドにアノテーションを付加することでユーザーの目的に沿った equals メソッドと hashCode メソッドを自動的に生成する手法を提案している [14]．この手法ではソースコードの動的解析を行っている．Grech らはソースコードを静的解析することによって，Rayside らの手法の問題点である循環したオブジェクト図の検査に時間がかかることを改善した [15]．Jensen らは clone メソッドによってオブジェクトのコピーを行うときの方針を示すアノテーションを提案している [16]．このアノテーションによって各クラスの clone メソッドではディープコピーを行うのかシャローコピーを行うのかを示し，統一した実装をサポートすることができる．

続いて Java の equals メソッドの検査に関する既存研究について説明する．equals メソッドの実装の検査手法として Rupakheti らによって提案されている手法 [9] が存在する．既 Java を対象とし

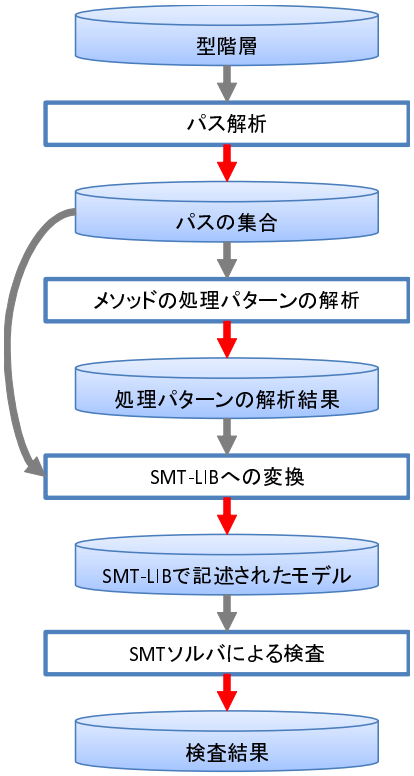

図 4: 先行研究の処理の流れ

た equals メソッドの検査を行っており，equals メソッドが満たすべき規則に違反しているかどうかを出力する．満たすべき規則に違反している場合は，その反例を Alloy Analyzer[17][18] の視覚的なインスタンス表示機能を用いて出力する．しかし，この既存手法には課題点が 2 つ存在する．1 つは equals メソッドをオーバーライドしているクラスがオーバーライドしなければならない hashCode メソッドの検査を行っていないことである．2 つ目は Alloy にはビット演算が存在しないために，ビット演算を使用する equals メソッドのモデル化が正しく行えておらず，正しい検査結果が得られないことである．

当研究グループではこの 2 つの問題を解決するための手法を過去に提案した．次節では過去に提案した手法について説明する．

2.4 先行研究

先行研究では Java コードの解析を行い，equals メソッドおよび hashCode メソッドの振舞いを SMT-LIB[19] を用いてモデル化し，SMT-LIB で記述されたモデルを SMT ソルバ [20][21][22][23][24] の 1 つである Z3[24] により検査する．

先行研究の処理の流れを図 4 に示す．円盤状のノードは資源を，四角のノードは処理を表している．赤い矢印は出力の流れを，黒い矢印は入力の流れを表している．先行研究は 1 つの型階層を入

力とし equals メソッドおよび hashCode メソッドがそれぞれの規則を満たしているかどうかを出力する．

図 4 に示す通り，先行研究は大きく 4 つのステップから成り立つ．パス解析では対象型階層中のソースコードの制御フローグラフ [25][26] を作成し，データフロー解析 [27] を行う．メソッドの処理パターンの解析では equals メソッドや hashCode メソッド中で行われる処理において，事前に定めておいた複数のパターンとのマッチングを行う．SMT-LIB への変換では処理のパターンの解析で得られた情報をもとにして SMT ソルバへの入力形式である SMT-LIB への変換を行う．SMT ソルバによる検査では SMT-LIB で記述されたモデルを SMT ソルバへ入力して検査を行い，性質を満たしているかどうかを出力する．性質満たしていない場合は反例も同時に出力する．

先行研究では equals メソッドの満たすべき性質については変更なく扱うが，hashCode メソッドの満たすべき性質については一部変更して扱っている．2.1.2 節で述べた通り，hashCode メソッドの満たすべき性質は以下の通りである．

- Java アプリケーションの実行中に同じオブジェクト上で複数回呼び出される場合は必ず，このオブジェクトに対する equals による比較で使われた情報が変更されていなければ，hashCode メソッドは同じ整数を一貫して返さなければならない．ただし，この整数は同じアプリケーションの実行ごとに同じである必要はない．

- equals(Object) メソッドで 2 つのオブジェクトが等価とされた場合，どちらのオブジェクトで hashCode メソッドを呼び出しても結果は同じ整数値にならなければならない．

- equals(java.lang.Object) メソッドで 2 つのオブジェクトが等価でないとされた場合は，これらのオブジェクトに対して hashCode メソッドを呼び出したときに，結果が異なる整数値にならなくてもかまわない．しかし，等しくないオブジェクトについては異なる整数値が生成されるようにすれば，ハッシュテーブルのパフォーマンスを上げることができる．

3 つ目の性質は，推奨されている性質であり必ずしも満たしている必要はないので先行研究では検査していない．2 つ目の性質は変更なくこのまま検査する．また，以降でこの性質を等価性と呼ぶ．

1 つ目の性質 (以降一貫性と呼ぶ) は時間の概念が含まれており SMT-LIB でモデル化することが困難なため，提案手法ではサブセット規則を用いて検査を行う．

また，[11] ではこの先行研究を既存手法である EQ[9] を拡張する形で実装ている．EQ には Eclipse 上の Java プロジェクトから equals メソッドに関する型階層を構築する処理があるので，equals メソッドだけではなく hashCode メソッドにのみ関係しているクラスも含むような型階層を構築するように拡張を行っている．また，パス解析やメソッドの処理パターンの解析の部分も EQ では equals メソッドのみを解析しているが，hashCode メソッドも解析するように拡張した．EQ における Alloy への変換と Alloy Analyzer による検査は，ツールではそれぞれ SMT-LIB への変換と SMT ソルバによる検査を行うようにした．配列や List，Set，Map に対する変換およびビット演算の変換は本ツール

において未実装であるため，型階層内でそれらのパターンが 1 つでも用いられていた場合，SMT-LIB への変換は行わず，SMT ソルバによる検査は行わない．ただし，フィールドのサブセット判定は可能であるため実行している．このツールに関しては、3 つのプロジェクト [28][29][30] に対して適用し評価を行っている．

2.4.1 サブセット規則

サブセット規則の定義は以下のとおりである．

- hashCode メソッドで参照されているフィールドの集合は，equals メソッドで参照されているフィールドのサブセットである．

これは，本来の一貫性における「情報」をフィールドに限定した場合に導き出される，本来よりも厳しい規則である．サブセット規則の具体例を図 5 に示す．図の equals メソッドでは isEqOnly，x，y の 3 つのフィールドを使用しているのに対し，hashCode メソッドでは x と y の 2 つのフィールドを使用している．このとき，hashCode メソッドの使用しているフィールドが equals メソッドの使用しているフィールドの部分集合であるため，この hashCode メソッドはサブセット規則を満たしている．

サブセット規則が成立することを subset，本来の性質である一貫性が成立することを base と表すと $base \Rightarrow subset$ という関係が成り立つが，$subset \Rightarrow base$ という関係は成り立たない．hashCode で参照されているフィールドが equals メソッドでで参照されているフィールドのサブセットであれば，一貫性は（「情報」をフィールドに限定した場合）必ず満たされるが，サブセットでない場合でも一貫性を満たしている場合がある．図 6 にその例を示す．この例ではコンストラクタで private int フィールドの hash を計算し，hashCode メソッドは hash の値を返すのみである．このクラスではコンストラクタと hashCode メソッド以外でフィールド hash の値は変更されておらず，また，hash は private であるため他クラスで勝手に値を変更されることもない．そのためこの実装は一貫性を満たしているといえるが，サブセット規則で見た場合 hash は equals メソッドで使用されていないため，この実装はサブセット規則を満たしていない．以後，一貫性を満たすがサブセット規則を満たさない場合を誤検出と呼ぶ．

```
public class IsSubset{
   public boolean equals(Object o){
      if(this.isEqOnly){
        this.x = 2;
        if(this.y != 0){
           …
        }
      }
   }
   public int hashCode(){
      return this.x + this.y * 16;
   }
}
```

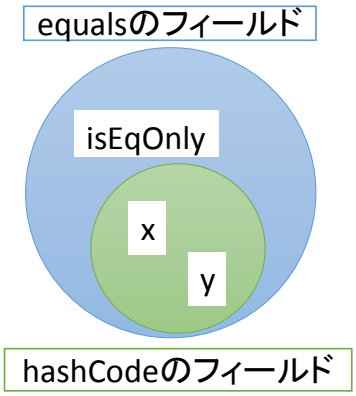

図 5: サブセット規則

```
private static final class ServiceKey{
  private final Bundle requesting;
  private final Bundle registering;
  private final Long serviceId;
  private final inthash;
  public ServiceKey(Bundle owningBundle, Bundle registeringBundle, Long property){
    requesting = owningBundle;
    registering = registeringBundle;
    serviceId= property;
    hash = serviceId.intValue() * 100003 + System.identityHashCode(requesting);
  }
  …
  public int hashCode(){
    return hash;
  }
  public boolean equals(Object other){
    if (other == this) return true;
    if (other == null) return false;
    if (other instanceofServiceKey) {
        ServiceKeyotherKey= (ServiceKey) other;
        return (otherKey.requesting == requesting
                                          && otherKey.serviceId.equals(serviceId));
    }
    return false;
  }
}
```

図 6: サブセットではないが，一貫性を満たす例

3　研究動機

本章では、本研究の動機について述べる．

当研究室の先行研究では，equals メソッドと hashCode メソッドがそれぞれ性質を満たしているかどうかを確かめるために SMT-LIB でモデル化し，SMT ソルバである Z3 を用いて検査する手法を提案し，その実装を行った．これにより equals メソッドと hashCode メソッドの性質を満たしてない実装を発見することができるようになった．しかし，モデル化の難しい一貫性に関してはサブセット規則をかわりに用いることで検証を行っている．しかし前章の最後に述べたように，サブセット規則は一貫性を厳しくした規則であり，サブセット規則で検出された違反の中には誤検出が存在する．サブセット規則違反の中において誤検出の割合が多くなると，見る必要のない hashCode メソッドにも目を通す必要が出てくるためバグの修正にかかるコストが高くなってしまう．

[11] では 3 つのオープンソースプロジェクトに対してツールを実行することで評価を行っているが，対象としたプロジェクトの規模は小さく，対象数も少ないため十分な評価を行えているとはいえない．また，eclipse プラグインである EQ[9] に拡張する形で実装を行っているため，先行研究における実装も eclipse のプラグインである．そのため，入力できるプロジェクトは eclipse 上で開発されたプロジェクトに限られるため，多数のプロジェクトを用意して評価を行うには不向きな実装となっている．

そのため，本研究では JDT を用いてサブセット規則を検査するツールを実装し，多数のプロジェクトに対してツールを実行してサブセット規則の精度に関する調査を行う．調査ではサブセット規則がどれくらい誤検出を起こしているか，誤検出のパターンにはどのようなものがあるのかに関して確認を行う．また，発見した誤検出のパターンに対してその改善方法の考察を行い，提案する．

4 実装

本章では調査を行うために実装したツールについて説明する．本実装では，使用しているフィールドの集合に追加する変数は，自クラスもしくは親クラスのフィールドか，他クラスの static なフィールドとしている．メソッド内で宣言されたローカル変数や仮引数は含めない．なお，追加する際は表 1 の規則に則ってのようにどのクラスのどういったフィールドかをわかりやすくする様にしている．

また，メソッド呼び出しがあった場合，そのメソッドの内部まで使用されているフィールドをチェックするが，static メソッドであるかそうでないかで処理が異なる．static なメソッドの場合は内部で使用されている変数は static フィールドのみが追加対象となる．さらに内部でのメソッド呼び出しに関しては，static なメソッドのみさらにその内部をチェックする．static ではないメソッドの場合は，ローカル変数と仮引数を除いたフィールドを集合に追加する．メソッド呼び出しは static メソッド，static ではないメソッドにかかわらず，その内部もチェックする．equals メソッドや hashCode メソッドをオーバーライドしていないクラスに関しては，直近の祖先クラスでオーバーライドされている equals メソッドや hashCode メソッドをそれぞれそのクラスのメソッドとする．祖先クラスにオーバーライドされておらず，java.lang.Object にいたった場合はそのメソッドではフィールドを使用していないものとして扱う．なお，equals メソッドと hashCode メソッドを両方とも実装していないクラスは今回検査の対象外であるため，祖先クラスは見ない．

実装は JDT を利用して行った．なお，本ツールでは [11] の実装とあわせて，表 2 を満たす equals メソッドと hashCode メソッドのみを対象としている．

実装の正しさに関しては，既存のツールの実験対象であった 3 つのプロジェクト Lucene4.6.0[28]，Tomcat8.0.1[30]，JFreeChart1.0.17[29] に対してツールを適用し，その出力を比較することで確認した．

その結果，既存のツールがサブセット規則に違反していないとする対象を違反であるとして出力

表 1: フィールド集合追加時の命名規則

フィールドの種類	static かどうか	登録名
自クラスもしくは親クラスのフィールド	static	完全限定名. 変数名
	static でない	this. 変数名
他クラスのフィールド	static	完全限定名. 変数名
	static でない	-

表 2: メソッドの検出条件

対象メソッド	メソッド名	引数の数	引数の型	返り値の型
equals メソッド	equals	1	java.lang.Object	boolean
hashCode メソッド	hashCode	0	-	int

としてしまう例はあるものの，既存のツールで違反とされるものを違反ではないと判断してしまうことはなかった．そのため，本ツールでサブセット規則に違反していないと判定された hashCode メソッドは一貫性を満たしているといえる．

上記の本ツールで対応できていない例に関しては 6 章にて詳細やその解決策について考察を行う．

5 調査

本章では，前章で説明したツールを用いて行った調査の詳細について記述する．

5.1 調査項目

本調査の調査項目は次の 2 つである．

- サブセット規則がどの程度の誤検出を引き起こすか
- 誤検出のパターンにはどのようなものがあるか

一つ目はサブセット規則の近似の精度を調査するために行う．サブセット規則で違反と検出されたものの中の誤検出の割合が低ければ低いほどサブセット規則の近似の精度は高いといえる．

二つ目は誤検出の改善方法を提案するための準備として行う．誤検出の原因となっている equals メソッドで使用されておらず hashCode メソッドで使用されているフィールドが，どういう用途で使用されているものなのかを調査しその目的ごとに分類する．

5.2 調査対象

今回，調査対象として Apache Software Foundation[12] で公開されている 130 のプロジェクトを対象として調査を行った．

また，プロジェクト内のすべてのクラスを検査の対象とするのではなく，equals メソッドと hashCode メソッドの少なくともどちらか 1 つをオーバーライドしているクラスを対象とする．このことに関して，図 7 を用いて説明する．Class A 及び Class C は equals メソッドと hashCode メソッドをどちらもオーバーライドされているため検査対象となる．Class B に関しては hashCode メソッドがオーバーライドされており，このクラスで equals メソッドを呼び出す場合は親クラスである Class A の equals メソッドが呼び出される．よってこのクラスをテストする際は Class B の hashCode メソッドと Class A の equals メソッドを対象としてサブセット規則による判定を行う．Class D に関しては，どちらのメソッドも呼び出されておらずこのクラスで呼び出される equals メソッドと hashCode メソッドは親クラスである Class C と同じものである．そのため，このクラスを対象として検査を行ったとしても親クラスとまったく同じ結果となるため，Class D は検査対象から除外される．

今回実験の対象にしたプロジェクトの LOC，総クラス数，実験対象のクラス数は付録 A に示す．

5.3 調査手順

本調査の手順は以下のとおりである．

1. 各プロジェクトに対してツールを実行し，サブセット規則違反を検出
2. もっとも多く違反の検出されたプロジェクトを目視で調査

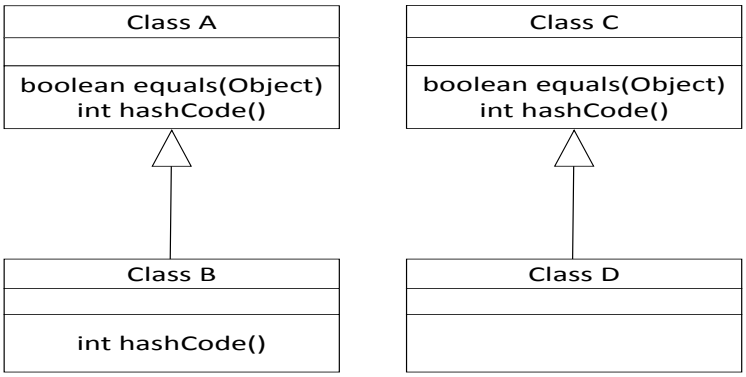

図 7: クラスと継承関係

3. 発見した誤検出の改善方法を考察し，ツールに実装

4. 改善を施したツールを用いて手順 1 から再度実行

図 8 はこの手順を図で示している．この手順を違反の多かったプロジェクトの出力の過半数が誤検出であるならば繰り返し実行する．誤検出ではない違反の割合が過半数に達した際は，すべてのプロジェクトの出力を調査する．

このような手順にした理由はサブセット規則の違反をすべて目視で確認するのはコストが高いと考えたためである．また，プロジェクトの実装方針によって定められた hashCode メソッドの実装が誤検出となる場合，そのプロジェクトには複数の誤検出が含まれている可能性が高く，その結果としてサブセット規則違反の数が多くなっているとも考えられるためである．

- サブセット規則の違反をすべて目視で確認するのはコストが高い考えたため
- 誤検出が原因で違反が多く検出される理由は一貫性に違反しているか誤検出が多いかのどちらかと考えたためである．

5.4 サブセット規則のみの適用結果

各プロジェクトから検出された違反数は付録 B に掲載しておく．130 のプロジェクトのうち，39 は違反数が 0 となった．一方，517 の検査対象クラスのうち 320 もの違反が見つかったプロジェクトもあった (hbase[31])．

表 3: 適用結果 (サブセット規則)

テスト対象のクラス数	サブセット規則違反のクラス数
11,764	1,608

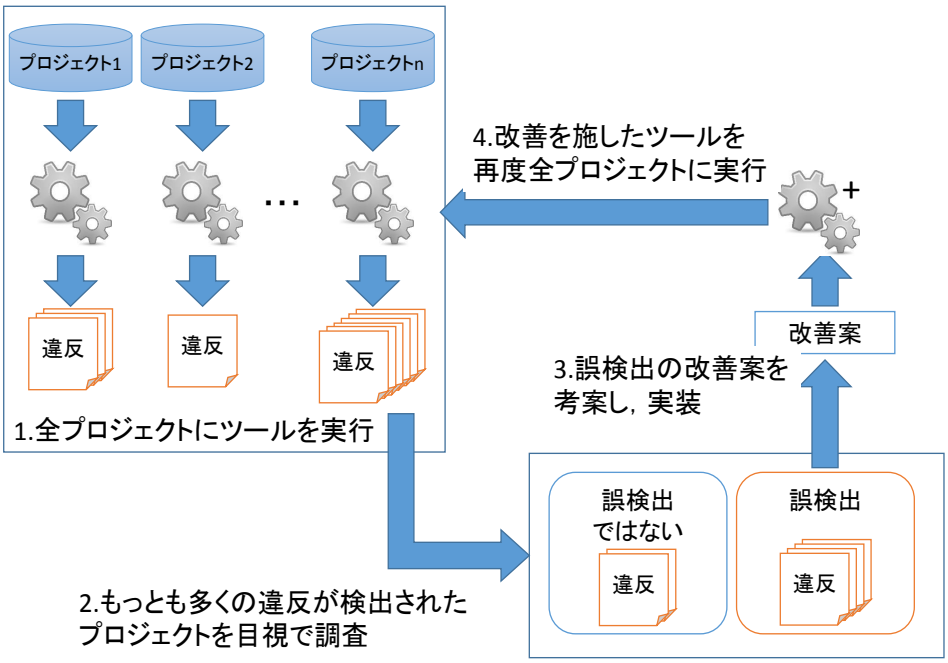

図 8: 調査手順

続いて，一番違反数の多かった hbase を対象として，出力が誤検出かどうかを目視で確認した．その結果，320 ある違反のうち 312 が誤検出であることがわかった．誤検出の原因となったフィールドに関して調査を行うと，以下の 2 つのパターンに分類することができた．

- hashCode メソッドの返す値をキャッシュに保存
- equals メソッドで使用されていないプリミティブ型の final フィールドを使用

一つ目のパターンの例を図 9 に示す．これは一度計算した hashCode メソッドの返り値をフィールドに保存し，2 回目以降の hashCode の呼び出しに対しては，そのキャッシュの値を返すといった実装である．この例では memoizedHashCode というフィールドに hashCode メソッドの返り値を保存しているが，このフィールドは equals メソッドで使用されていないためサブセット規則違反として検出される．このフィールドは hashCode 以外で勝手に値を変更させられなければ常に変わらぬ値を返し続けるため，一貫性に違反していない．ただし，コンストラクタに関してはオブジェクト生成時に一度しか呼ばれないため，コンストラクタ何度も呼び出されることで値が勝手に変わるという状況は起こりえないので，コンストラクタでの使用は問題ないといえる．また，public や protected，アクセス修飾子なしのフィールドに関しては他のクラスから値を変更することが可能であるため，private であるフィールドのみが今回の対象となる．

```
public static final class GrantResponse extends com.google.protobuf.GeneratedMessage
                                        implements GrantResponseOrBuilder {
    …
    @java.lang.Override
    public boolean equals(final java.lang.Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof GrantResponse)) return super.equals(obj);
        GrantResponse other = (GrantResponse) obj;
        boolean result = true;
        result = result && getUnknownFields().equals(other.getUnknownFields());
        return result;
    }
    private int memoizedHashCode = 0;
    @java.lang.Override
    public int hashCode() {
      if (memoizedHashCode != 0) return memoizedHashCode;
      int hash = 41;
      hash = (19 * hash) + getDescriptorForType().hashCode();
      hash = (29 * hash) + getUnknownFields().hashCode();
      memoizedHashCode = hash;
      return hash;
    }
}
```

図 9: ハッシュコードをキャッシュに保存

二つ目のパターンの例を図 10 に示す．これは equals メソッドで使用されていない private static final int 型のフィールドを hashCode メソッドで使用しているためにサブセット規則違反として検出される．しかし，プリミティブ型の再代入不可であるフィールドは実行中に値を変更することが不可能なため，このフィールドが原因で hashCode メソッドの返り値が変わることはない．同様に String や Enum といったような不変クラスの型においてもそのフィールドが再代入不可で宣言されているならば，一度値が決まると変わることがない．そのため，これらのフィールドに関しては equals メソッドで使用されていなくても hashCode メソッドで使用可能だと判断できる．

以上 2 つの誤検出のパターンの改善策として，サブセット規則の基準を緩和する以下の緩和条件を提案する．

緩和条件 1 hashCode メソッドとコンストラクタ以外で使用されていないフィールドは equals メソッドでの使用の有無にかかわらず hashCode で使用可能

緩和条件 2 不変クラスを型にもち，かつ再代入不可であるフィールドは equals メソッドでの使用の有無にかかわらず hashCode メソッドで使用可能

ただし，実装の都合上緩和条件 1 は private であるフィールドに限定することで，他のクラスの確認を行わない．緩和条件 2 に関しても，すべての不変クラスの型に対応できておらず，プリミティブ型と String，Enum のみに対応している．この 2 つの緩和条件をツールに実装し，再度すべてのプロジェクトに対して実行した．

```
public static final class CountResponse extends com.google.protobuf.GeneratedMessage
                                        implements CountResponseOrBuilder {
    public static final int COUNT_FIELD_NUMBER = 1;
    …
    @java.lang.Override
    public boolean equals(final java.lang.Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof CountResponse)) return super.equals(obj);
        CountResponse other = (CountResponse) obj;
        boolean result = true;
        result = result && (hasCount() == other.hasCount());
        if (hasCount()) result = result && (getCount() == other.getCount());
        result = result && getUnknownFields().equals(other.getUnknownFields());
        return result;
    }
    @java.lang.Override
    public int hashCode() {
      int hash = 41;
      hash = (19 * hash) + getDescriptorForType().hashCode();
      if (hasCount()) {
        hash = (37 * hash) + COUNT_FIELD_NUMBER;
        hash = (53 * hash) + hashLong(getCount());
      }
      hash = (29 * hash) + getUnknownFields().hashCode();
      return hash;
    }
```

図 10: プリミティブ型の final フィールドを使用

5.5 緩和条件追加後の適用結果

緩和条件追加後の違反数を表 4 に示す．緩和条件 1 を追加することで 638 個，緩和条件 2 を追加することで 369 個，両緩和条件を追加することで 1,079 個の誤検出を削減することに成功した．また，130 のプロジェクトのうち，54 のプロジェクトが違反数が 0 となり，違反数が一番多いプロジェクトに関しても 59 となった．

もっとも違反数の多い directory を対象として，出力が誤検出かどうかを再度目視で確認した．その結果，59 のうち 54 が一貫性にも違反しており，誤検出がそこまで生じていないことがわかる．この結果から，全プロジェクトへのツールの実行により得られた 529 の違反すべての目視での確認を行った．

529 のうち，一貫性を満たしていない，すなわち本来検出したかった違反の数は 394，誤検出は 135 となった．次節では，この 135 の誤検出をさらに詳しく見ていく．

表 4: 適用結果 (緩和条件追加後)

サブセット規則のみ	1,608
緩和条件 1 のみ追加	970
緩和条件 2 のみ追加	1,239
両緩和条件を追加	529

5.6 誤検出の分類

135 の誤検出をさらに詳しく見て，誤検出の原因となっている変数や実装を以下の 9 つに分類した．

1. オーバーライドした equals メソッドで this==obj のみで判定
2. equals(Object) をオーバーライドするのではなく，クラスを引数として受け取る equals メソッドを定義
3. hashCode メソッドの返り値を計算するメソッドの実装
4. hashCode メソッドの値を計算しなおすかどうかをフラグを用いて管理
5. hashCode メソッドの返り値をキャッシュとして保存しているフィールドを hashCode 以外のメソッドで参照としてのみ利用
6. メソッドの呼出回数を計算する InvocationCounter クラス
7. デバッグ用のフラグを hashCode メソッドのみで使用
8. abstract クラスで hashCode が実装されており，equals メソッドが abstract
9. 実装の不備による誤検出

これら 9 つの分類に関する詳しい説明及び解決方法に関しては 6 章で述べる．

```
public class Modes implements java.io.Serializable {
    private String _value_;
    …
    public boolean equals(Object obj) {
        return (obj == this);
    }
    public int hashCode() {
        return toString().hashCode();
    }
    public String toString() {
        return _value_;
    }
}
```

図 11: this==obj のみの判定

6 考察

本章では 5 章の結果を受け，その考察を行っている．

6.1 サブセット規則の近似の精度について

調査の結果から，サブセット規則のみの検査で十分であったプロジェクトは 130 中 44 であった．過半数のプロジェクトにおいては，差はあるもののその出力の中には誤検出が含まれていた．また，hbase のように，517 分の 320 という過半数が違反であると出力されたにもかかわらず，一貫性を満たしていないものはそのうちの 6 つというようにプロジェクトによってはほとんどが誤検出である場合がある．

全 130 プロジェクトの合計で見てもサブセット規則違反と判定されるのが 1,608 あるのに対し，実際の一貫性違反は 394 であり精度が高いとは言い切れない．

6.2 誤検出とその改善方法について

この節では，5 章で分類した誤検出の各パターンに対しての考察とその解決策について述べる．

6.2.1 this==obj のみを判定する equals メソッド

誤検出で一番多かったのが，図 11 のようなオーバーライドした equals メソッドで this==obj のみを判定しているものである．これは equals メソッドではフィールドを使用しておらず，hashCode メソッドは値の計算にフィールドを使用しているためにサブセット規則違反となる．しかし，この場合の equals の情報を this だとすると，this が変わらない限りフィールドの値も変わりようがないため，一貫性は満たしているといえる．

このパターンの改善策としては，equals メソッドが this==obj の判定のみを行っている場合はすべてのフィールドを保持しているとすれば，この誤検出を改善することが可能である．

```
public class BooleanResponse implements CacheResponse {
    protected boolean _bool = false;
    …
    public boolean getValue() { return _bool; }
    public boolean equals(BooleanResponse req) {
        try {
            return (_bool == req.getValue());
        } catch(NullPointerException e) {
            return false;
        }
    }
    public int hashCode() { return (_bool ? 1 : -1); }
}
```

図 12: クラスを引数にとる equals

6.2.2 クラスを引数にとる equals メソッドの実装

二番目に多かったのが，equals(Object) をオーバーライドするのではなく equals メソッドの宣言されているクラスを引数にとるような equals メソッドを定義している場合である．このパターンが違反として検出される理由としては，表 2 で示しているとおり，今回対象としている equals メソッドが equals(Object) であるためにこの equals メソッドを無視しているからである．

このパターンの改善策は，同じクラスを引数にとる equals メソッドも対象に含めることである．ただし，equals(Object) も同じく定義されている場合はこちらと equals メソッドを対象とするといったような順位付けが必要になる．

また，満たすべき性質としては equals(Object) で等価と判定される場合といった記述であるため，仮に性質を満たしていたとしても，equals(Object) メソッドも宣言すべきであるという注意をする必要もあるかもしれない．

6.2.3 ハッシュ値計算用メソッドを実装

図 13 のように hashCode メソッドは hashCode メソッドの返り値計算用のメソッドを呼び出すだけで，実際の計算は別メソッドで行われているといった例も存在する．この場合，hashCode メソッドではキャッシュ用のフィールドのみを使用しており，またそのキャッシュ用のフィールドは計算用のメソッドでも使用されているためサブセット規則及び両緩和規則でも規則違反となる．改善案としては，他メソッドで定義されている場合，その値は equals メソッドで使用されているフィールドのみから計算されているという条件を用いればよい．代入文が複数存在する場合は，すべて同じ計算式であるかどうかも確認する必要がある．

```
class MemberTableHash {
    String name; String descriptor;
    int   index; int   hashCode;

    void setHashCode() {
        hashCode = name.hashCode() + descriptor.hashCode();
    }
    public boolean equals(Object other) {
        MemberTableHash mth = (MemberTableHash) other;
        if (other == null) return false;
        if (name.equals(mth.name) && descriptor.equals(mth.descriptor)) {
            return true;
        } else {
            return false;
        }
    }
    public int hashCode() {
        return hashCode;
    }
}
```

図 13: ハッシュ値計算用メソッドの実装

6.2.4 フラグによるハッシュ値の計算判定

図 14 のように，再度 hashCode メソッドの返り値を計算するかどうかをフラグで管理しているものがある．このフラグは equals メソッドで使用されている情報が変更された場合に false となり，再度 hashCode メソッドの返り値を計算するといったものである．このフラグが外部から勝手に変更される可能性があるならば違反であるが，外部から変更される可能性がなく，また，必ず equals メソッドの情報が変更された場合のみ変更されているならば，一貫性は満たしている．

そのためこの誤検出の改善方法としては，フラグが private なフィールドであり，かつ equals メソッドで使用されているフィールドが変更された場合のみ変更されているならば hashCode メソッドで使用可能であるといえる．

6.2.5 キャッシュ用変数を他メソッドで参照で使用

緩和条件 1 により，hashCode メソッドの返り値をキャッシュとしてもつフィールドは private でかつ hashCode 及びコンストラクタ以外では使用不可としていた．しかし，他メソッドでの使用が参照であるならばその値が変更されることはないため，問題がないといえる．

そのため，緩和条件 1 では他メソッドでの使用を禁止しているが，そうではなく他メソッドでの定義が行われていなければ，equals メソッドで使用されていなくても hashCode メソッドで使用可能とすればよい．

```
public class GenericEntity implements Map<String, Object>, LocalizedMap<Object>,
                                Serializable, Comparable<GenericEntity>, Cloneable {
    private boolean generateHashCode = true;
    private int cachedHashCode = 0;
    …
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof GenericEntity)) return false;
        try {
            return this.compareTo((GenericEntity) obj) == 0;
        } catch (ClassCastException e) {
            return false;
        }
    }
    @Override
    public int hashCode() {
        if (generateHashCode) {
            cachedHashCode = 0;
            if (getEntityName() != null) {
                cachedHashCode += getEntityName().hashCode() >> 1;
            }
            cachedHashCode += fields.hashCode() >> 1;
            generateHashCode = false;
        }
        return cachedHashCode;
    }
}
```

図 14: フラグを用いた計算判定

6.2.6 InvocationCounter クラス

特殊な実装の例として図 15 のようなメソッドの呼出回数を計算する InvocationCounter というクラスが存在している．これは名前のとおり，メソッドが呼ばれるたびにカウントを 1 増やし，既定値を返すといった処理を行っている．

この実装は非常に珍しいものであり，また，通常の実装とは異なるため無理に対応させる必要はないと考える．

6.2.7 デバッグ用フラグの使用

hashCode メソッドのみでデバッグ用のフラグを使用しているためにサブセット規則違反として検出された例も存在した．今回デバッグ用フラグを使用していたのは catch ブロック内のみであった．catch ブロックは何らかの例外が生じた場合のみ実行される例外処理の部分になるので，通常使用されることはない．

そのためこのパターンに対応する場合は catch ブロック内で使用されている変数はフィールドの集合に含めないとすればよい．

```
 static class InvocationCounter implements LongCollection {
    private int _equals;
    private int _toString;
    private int _hashCode;
    …
    public boolean equals(Object obj) { _equals++; return false; }
    public int hashCode() { _hashCode++; return 0; }
}
```

図 15: InvocationCounter

```
public final int hashCode() {
    try {
    if (getValue() == null)
        return 0;
    } catch (StandardException se) {
        if (SanityManager.DEBUG)
            SanityManager.THROWASSERT("Unexpected exception", se);
        return 0;
    }
    byte[] bytes = dataValue;
    int lastNonPadByte = bytes.length - 1;
    while (lastNonPadByte >= 0 && bytes[lastNonPadByte] == PAD) {
        lastNonPadByte--;
    }
    int hashcode = 0;
    for (int i = 0; i <= lastNonPadByte; i++) {
        hashcode = hashcode * 31 + bytes[i];
    }
}
```

図 16: デバッグ用フラグの使用

6.2.8 equals メソッドが abstract クラス

このような実装は abstract クラスでしか現れない実装である．図 17 のように hashCode メソッドに関しては実装されているものの，equals メソッドが abstract メソッドになっているため，サブセット規則違反が生じる．しかし，これは実装上の都合のため，実際に背反しているとはいえない．

このパターンに対応しようとする場合，equals メソッドが abstract メソッドであった場合はこのクラスを対象から除くという処理を行えばよい．この例とは逆に equals メソッドが実装されていて hashCode メソッドが abstract の場合は hashCode メソッドはフィールドを使用していないと判定され，equals メソッドの使用しているフィールドの集合にかかわらずサブセット規則を満たすと判定されるため，特に考慮する必要はない．

```java
public abstract class Node {
    final protected Object label;
    static final int THRESHOLD = 10000;
    …
    @Override
    public abstract boolean equals(Object o);
    @Override
    public int hashCode() { return label.hashCode() * 31; }
}
```

図 17: equals の実装が abstract

```java
public class OpenBitSet extends DocIdSet implements Bits, Cloneable {
    protected long[] bits; protected int wlen; private long numBits;
    …
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof OpenBitSet)) return false;
        OpenBitSet a;
        OpenBitSet b = (OpenBitSet)o;
        if (b.wlen > this.wlen) { a = b; b=this; }
        else { a=this; }
        for (int i=a.wlen-1; i>=b.wlen; i--) { if (a.bits[i]!=0) return false; }
        for (int i=b.wlen-1; i>=0; i--) { if (a.bits[i] != b.bits[i]) return false; }
        return true;
    }
    @Override
    public int hashCode() {
        long h = 0;
        for (int i = bits.length; --i>=0;) {
            h ^= bits[i];
            h = (h << 1) | (h >>> 63);
        }
        return (int)((h>>32) ^ h) + 0x98761234;
    }
}
```

図 18: 実装都合による誤検出

6.2.9 実装都合による誤検出

実装都合による誤検出は図 18 のような例である．これは equals メソッドで this をローカル変数に代入しており，そのローカル変数からフィールドにアクセスしているため，そのフィールドを使用したフィールドとして追加することができていない．

このパターンに対応するには，this が代入されている場合，this とその変数と対応関係をもたせ，変数がフィールドを呼び出した際，その変数が this と対応関係を持っているならばフィールドが呼び出されたとするといった処理が必要になる．

6.3 一貫性を満たさない hashCode メソッドの傾向

ここでは，違反調査の結果わかった一貫性を満たさない hashCode メソッドの傾向について述べる．特に多かった傾向としては以下の 2 つが主である．

- 一貫性を満たしていない親クラスをもつクラスの hashCode メソッドで super.hashCode() を呼出
- hashCode メソッドのみがオーバーライドされ，自クラスのフィールドを使用

1 つ目は子クラスは修正する必要がなく，親クラスを修正する必要がある例である．directory というプロジェクトに特に多く，59 の違反出力のうち，29 がこの理由のための出力であった．違反であることにかわりはないため出力されること自体に問題はないが，親クラスを優先的に見ることができればすべての違反を見る必要がなくなる．

2 つ目に関しては，明らかに一貫性に違反する実装であるため，ユーザはすぐに修正する必要があると考えられる．

そのため，これらの情報を利用して，親クラスも違反している場合は親クラスを優先して確認するように，hashCode のみオーバーライドされている場合も実装上問題があるために修正する必要があるといったような，違反確認の優先順位を示すような実装も考えられる．

7　あとがき

当研究室では先行研究として hashCode メソッドの整合性を検査する手法を提案し，SMT-LIB でのモデル化の難しい一貫性に関してはサブセット規則を用いて検証を行っていた．しかし，サブセット規則は一貫性よりも制約が厳しいために誤検出が起こる可能性がある．

本研究では，サブセット規則による検査を行うツールを実装し，Apache Software Foundation で公開されている 130 のプロジェクトを対象としてサブセット規則で起こる誤検出に関する調査を行い，サブセット規則の近似の精度を確認した．また誤検出をパターン分けし，その改善方法の考察，提案を行った．提案した改善方法のうち 2 つに関しては実装を行い，その適用により減少する誤検出の数を調査した．結果としてサブセット規則やその精度の改善方法に関する知見を得ることができた．

今後の課題としては，今回提案し，まだ実装できていない改善方法をツールに実装し，その評価を行うことが考えられる．また，考察で述べたように一貫性に違反しているものに関して優先順位をつけることが可能であると考えるため，その手法についてもツールを実装することで自動化することが可能と考えられる．また，先行研究におけるツールを組み合わせることで equals メソッドと hashCode メソッドを高い精度で検査できるツールの作成も行っていきたい．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究の全過程を通して， 丁寧なご指導を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に関して，厳しくも的確なご助言を頂きました 井垣 宏 特任准教授 に深く感謝申し上げます.

本研究を行うにあたり，日常の議論の中で有益かつ的確なご助言を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究に用いたツールの実装に際して熱心にご指導いただき，また本研究に関して多大なるご助言，ご助力をいただきました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻研究員の堀田 圭佑 氏に深く感謝申し上げます.

その他の楠本研究室の皆様のご助言，ご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等で多くの知識や示唆を頂きました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] Joshua Bloch. *"Effective Java"*. Addison-Wesley, 2008.

[2] Oracle. "Java Platform, Standard Edition 8 API Specification", 2015. http://docs.oracle.com/javase/8/docs/api/.

[3] David Hovemeyer and William Pugh. "Finding bugs is easy". In *ACM SIGPLAN Notices Homepage archive*, pp. 92–106, 2004.

[4] Julian Dolby, Mandana Vaziri, and Frank Tip. "Finding bugs efficiently with a SAT solver". In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 195–204, 2007.

[5] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. "Declarative Object Identity Using Relation Types". In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pp. 54–78, 2007.

[6] Chandan R. Rupakheti and Daqing Hou. "An Empirical Study of the Design and Implementation of Object Equality in Java". In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pp. 111–125, 2008.

[7] Chandan R. Rupakheti and Daqing Hou. "An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java". In *Proceedings of the 17th Working Conference on Reverse Engineering*, pp. 205–214, 2010.

[8] Chandan R. Rupakheti and Daqing Hou. "EQ: Checking the Implementation of Equality in Java". In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pp. 590–593, 2011.

[9] Chandan R. Rupakheti and Daqing Hou. "Finding Errors from Reverse-Engineered Equality Models using a Constraint Solver". In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 77–86, 2012.

[10] Hiroaki Shimba, Takafumi Ohta, Hiroki Onoue, Kozo Okano, and Shinji Kusumoto. Formal verification technique for consistency checking between equals and hashcode methods in java. In *IWIN2014*, 9 2014.

[11] 尾ノ上博樹. "Java における equals メソッドと hashCode メソッドの検査ツールの実装と評価", 2014.

[12] Apache. “The Apache Software Foundation”, 2015. http://www.apache.org/.

[13] the Eclipse Foundation. “ Eclipse Java development tools (JDT)”, 2013. http://www.eclipse.org/jdt/.

[14] Derek Rayside, Zev Benjamin, Rishabh Singh, Joseph P. Near, Aleksandar Milicevic, and Daniel Jackson. “Equality and Hashing for (almost) Free: Generating Implementations from Abstraction Functions”. In *Proceedings of the 31st International Conference on Software Engineering,*, pp. 342–352, 2009.

[15] Neville Grech, Julian Rathke, and Bernd Fischer. “JEqualityGen: Generating Equality and Hashing Methods”. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pp. 177–186, 2010.

[16] Thomas Jensen, Florent Kirchner, David Pichardie, and Inria Rennes Bretagne Atlantique. “Secure the clones: Static enforcement of policies for secure object copying”. Technical report, 2010.

[17] Daniel Jackson. “Alloy: a lightweight object modelling notation”. *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 2, pp. 256–290, 2002.

[18] Daniel Jackson. “ Alloy: a language and tool for relational models”, 2013. http://alloy.mit.edu.

[19] Clark Barrett, Aaron Stump, and Cesare Tinelli. “The SMT-LIB Standard Version 2.0”, 2012. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf.

[20] Bruno Dutertre and Leonardo de Moura. “A Fast Linear-Arithmetic Solver for DPLL(T)”. In *Proceedings of the 18th international conference on Computer Aided Verification*, pp. 81–94, 2006.

[21] Clark Barrett and Cesare Tinelli. “CVC3”. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pp. 298–302, 2007.

[22] Alberto Griggio. “A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic”. *JSAT*, Vol. 8, pp. 1–27, 2012.

[23] Alessandro Cimatti, Alberto Griggio, Bastiaan J. Schaafsma, and Roberto Sebastiani. “ The MathSAT5 SMT Solver”. In *Proceedings of the 19th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 93–107, 2013.

[24] Leonardo de Moura and Nikolaj Bjorner. "Z3: An Efficient SMT Solver". In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 337–340, 2008.

[25] James Stanier and Des Watson. "Intermediate representations in imperative compilers : A survey". In *ACM Computing Surveys*, p. 127, 2013.

[26] Frances E. Allen. "Control flow analysis". In *Proceedings of a symposium on Compiler optimization*, p. 119, 1970.

[27] John B. Kam and Jeffrey D. Ullman. "Global data flow analysis and iterative algorithms". In *Journal of the ACM*, p. 158176, 1976.

[28] Apache. "Apache Lucene - A Lucene Core", 2012. http://lucene.apache.org/core/.

[29] Object Refinery Limited. "JFreeChart", 2012. http://www.jfree.org/jfreechart/.

[30] Apache. "Apache Tomcat - Welcome", 2014. http://tomcat.apache.org/.

[31] Apache. "HBase Apache HBase", 2015. http://hbase.apache.org/.

付録 A

実験対象プロジェクトの規模

ここでは，今回の実験に用いた全プロジェクトの規模を示す．全プロジェクトを LOC で昇順に並べている．

プロジェクト名	LOC	総クラス数	検査対象のクラス数	プロジェクト名	LOC	総クラス数	検査対象のクラス数
infrastructure	239	3	0	shale	122,380	742	17
lucy	240	1	0	santuario	123,417	805	26
trafficserver	544	7	0	labs	136,471	1,510	35
buildr	562	10	0	hivemind	138,690	899	11
esme	839	7	0	ode	145,359	1,458	34
gump	2,496	36	1	karaf	149,814	1,625	28
mesos	2,829	29	2	click	153,172	813	12
kafka	3,073	28	1	continuum	155,935	827	6
libcloud	3,977	48	0	pivot	163,050	1,474	21
mrunit	10,704	89	3	shindig	165,696	1,297	38
thrift	23,909	205	1	syncope	168,704	1,257	10
oltu	26,333	308	5	rave	169,377	1,391	94
whirr	27,124	274	8	cassandra	172,737	1,055	127
creadur	27,879	287	10	excalibur	183,347	1,047	12
gora	31,229	204	15	mahout	186,227	1,449	58
forrest	32,491	246	0	sis	188,158	1,004	61
subversion	43,696	258	14	turbine	189,561	1,026	8
onami	44,391	682	13	jmeter	190,334	1,173	16
any23	47,501	405	7	pdfbox	196,674	1,041	17
wookie	48,871	299	3	ibatis	197,434	1,344	28
flume	48,986	405	6	juddi	197,522	986	10
directmemory	49,051	467	53	abdera	209,275	1,715	89
nutch	56,348	427	12	oodt	210,490	1,674	49
chukwa	59,422	526	7	xmlbeans	233,687	1,386	20
giraph	60,814	574	5	zookeeper	238,021	1,997	50
sqoop	64,162	521	4	archiva	251,352	1,561	57
empire-db	67,691	439	7	velocity	253,615	890	13
hama	69,533	599	24	stanbol	259,627	1,744	82
bval	70,905	836	19	airavata	272,177	1,873	39
tiles	75,793	647	10	ace	272,196	899	23
tika	81,256	591	9	logging	272,244	1,946	22
openmeetings	82,113	539	3	ctakes	299,962	1,673	25
avro	85,402	913	62	httpcomponents	317,338	1,976	34
jspwiki	99,807	487	18	xerces	321,531	1,562	31
clerezza	105,046	823	51	beehive	345,682	3,577	48
roller	107,497	649	38	ofbiz	387,408	2,085	151
openwebbeans	110,019	1,315	11	cayenne	394,069	3,505	30
shiro	113,123	932	22	mina	396,232	2,802	44
xml	116,636	765	6	lenya	402,391	1,193	21
opennlp	119,889	1,186	25	xalan	407,948	1,314	16

図 19: 実験対象プロジェクトの規模 (1/2)

プロジェクト名	LOC	総クラス数	検査対象のクラス数	プロジェクト名	LOC	総クラス数	検査対象のクラス数
synapse	410,747	1,918	6	axis	1,408,900	7,946	133
pig	414,885	2,667	59	activemq	1,426,529	7,119	283
aries	421,019	2,227	81	maven	1,471,625	9,276	116
chemistry	425,722	2,216	10	openoffice	1,479,567	5,182	43
servicemix	472,140	3,163	29	webservices	1,545,600	9,596	170
ant	494,588	3,510	84	jackrabbit	1,575,197	7,100	261
poi	522,096	3,078	94	manifoldcf	1,729,831	5,975	226
jakarta	549,233	3,348	75	james	1,734,893	4,584	103
qpid	562,436	4,071	90	myfaces	1,744,930	10,236	274
avalon	562,711	2,655	47	jena	2,051,402	10,352	362
wink	604,835	3,836	64	commons	2,436,704	14,768	621
struts	622,433	4,418	84	tomee	2,682,827	12,011	507
tapestry	627,659	5,298	62	db	2,713,245	8,166	487
portals	656,516	4,015	192	cxf	3,097,081	7,852	87
sling	671,840	3,225	57	directory	3,611,460	12,314	397
etch	704,989	1,055	12	openjpa	3,731,537	5,333	417
hive	731,376	5,602	467	incubator	3,878,405	22,845	390
cocoon	763,580	4,549	70	harmony	4,076,608	20,131	538
hbase	777,457	4,592	517	flex	4,262,093	7,368	171
xmlgraphics	780,534	5,610	79	tomcat	4,383,769	5,367	81
uima	889,226	5,472	93	tuscany	5,119,257	10,723	147
oozie	924,394	1,185	8	geronimo	5,488,196	14,994	310
lucene	956,983	6,607	312	felix	5,561,075	8,406	202
hadoop	1,054,631	7,818	264	river	8,630,783	5,571	370
camel	1,123,483	10,360	58	合計	105,003,060	423,367	11,764
isis	1,174,309	10,995	166				

図 20: 実験対象プロジェクトの規模 (2/2)

付録 B

ツールの実行によって得られた違反数

サブセット規則による検査および緩和条件を追加した規則による検査で違反数が 0 だったプロジェクトは以下のとおり．これらのプロジェクトに関しては図から省略している．

airavata, any23, buildr, click, creadur, directmemory, empire-db, esme, etch, flume, forrest, giraph, gump, infrastructure, juddi, karaf, lenya, libcloud, lucy, mahout, mesos, mrunit, nutch, onami, openmeetings, opennlp, pivot, roller, shale, subversion, synapse, syncope, thrift, tika, tiles, rafficserver, turbine, velocity, wookie

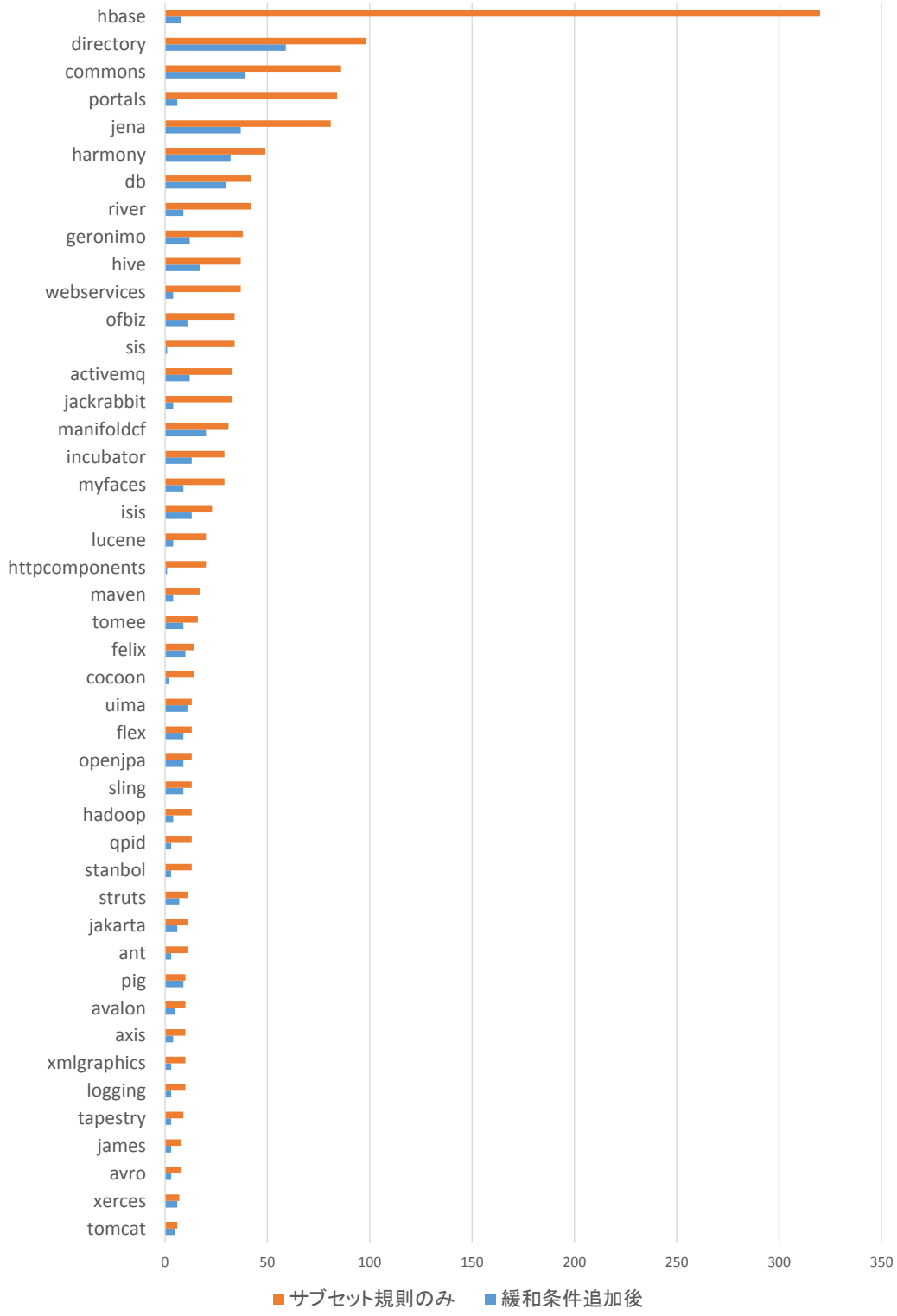

図 21: 緩和条件追加前後の違反数の推移 (上位)

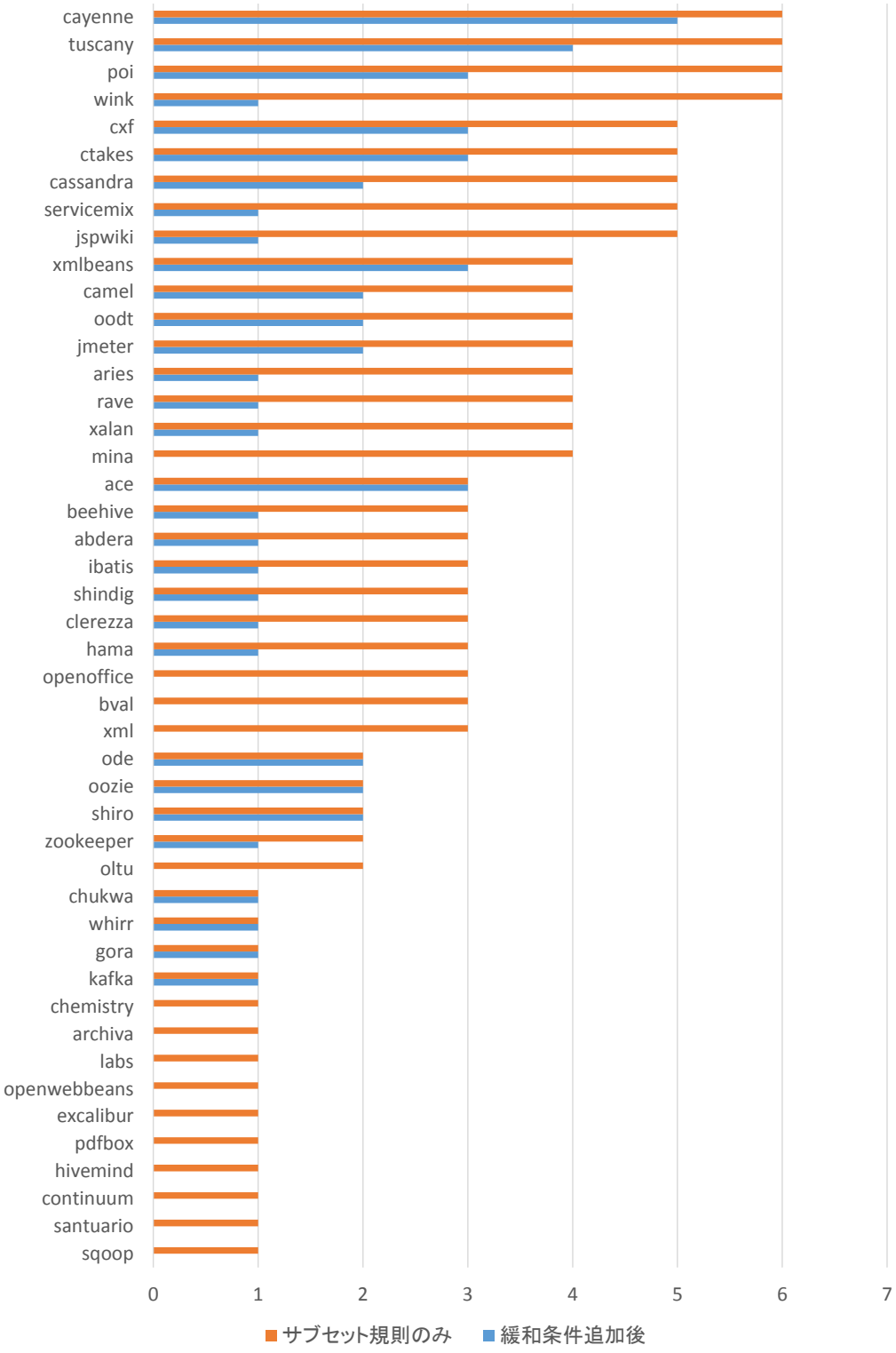

図 22: 緩和条件追加前後の違反数の推移 (下位)