Technical Report

ソースコード縮退によるソースコード理解

神谷 年洋

March 15, 2004

Abstract

本稿では，ソースコードを理解するための方法の一つとして，ソースコードの規模を擬似的に小さくすることでソースコードのレビューを容易にする手法を提案する．提案する手法は，ソースコードに相互に依存する部品群が含まれる場合でも，その一部を取り出して調べることが可能である．さらに，本手法を利用してインクリメンタルにソースコード理解を行うことができる．ケーススタディにより具体的な適用例を示すとともに，本手法の有効性を評価する．

1 はじめに

ソフトウェアの保守作業においては，不具合の原因を特定してソースコードを修正する，機能拡張のための修正を行う，処理上のボトルネックやセキュリティホールといった潜在的な欠陥を調べるなどのために，保守作業者がソースコードを理解することが必要とされる．

ソースコードを理解する方法の一つに，ソフトウェアに関する文書を参照することがある．仕様や設計を理解することで，ソースコードを理解する手がかりを得ることができる．もう一つの方法としては，オブジェクトコードやソースコードから情報を抽出する手法がある．

ソースコードの理解を目的として，ソースコードを分析する手法としては，(1) リバースエンジニアリング，すなわち，ソースコードから抽象度の高い情報を生成する手法 (ソースコードから Unified Modeling Language のクラス図やコラボレーション図を生成するツール)，(2) ソースコードから，何らかの意味的なまとまりを抽出する手法 (プログラムスライシング [8, 11] や Concept Lattice[1])，(3) ソースコードには明示的に記述されない情報を解析する手法がある (クロスリファレンス，ソフトウェア部品の分類 [12], エイリアス解析 [6], コードクローン分析 [2] など)．

この中でも，ソースコードから何らかの意味的なまとまりを抽出するプログラムスライシングや Concept Lattice といった手法では，ソフトウェアの適切な構成要素 (手法に依存して，クラスやソースコード中の文) の依存関係を追跡することにより，ある要素に関係のあるすべての要素を抜き出すアプローチを取って

いる．要素間に相互依存関係がある場合には，それらの要素は併合されるか，依存関係のあるすべての要素が取り出されることになる．現実に，大規模なソフトウェアの構成要素が複雑な依存関係を持つ例が報告されており [9]，これらの手法を効果的に適用するためには，何らかの方策が必要である．

本稿では，ソースコードを理解するための分析手法の一つとして，ソースコードの規模を擬似的に小さくする手法である「ソースコード縮退」を提案する．この手法は，構成要素の間に依存関係がある場合に，一定の基準によってそのような依存関係を切り離すことで，ソフトウェアの規模を見かけ上小さくする．ソースコード縮退をコードレビューに応用すれば，相互依存関係を持った要素群が含まれる場合でも，少数の要素を取り出してレビューすることが可能になり，ソースコードの理解が容易になる．

2 ソースコード縮退

2.1 ソースコード縮退とは

ソースコード縮退 (degeneracy) とは，ソフトウェアを構成する要素 (後述のケーススタディでは，クラスやコンポーネントが要素となる) 群のうち，一部の要素を副作用のない擬似的な定数 (以下,「擬似定数要素」) と見なし，関連するソースコードを除去する．具体的には，以下のような部分が除去される．

- 擬似定数要素のソースコード
  擬似定数要素自体を定義する部分を削除することができる．
- 擬似定数要素への参照
  例えば，擬似定数要素による条件分岐 (if 文や throw 文) は，定数による分岐であるため，分岐の一方のみに制御が移ると仮定し，それ以外の部分のソースコードを取り除くことができる．
- 擬似定数要素を計算結果とするルーチン
  例えば，擬似定数要素を戻り値とし，副作用を持たない手続き (メソッド) は取り除くことができる．
- 擬似定数要素を格納するデータ構造
  例えば，擬似定数要素のみを格納する配列は取り除くことができる．

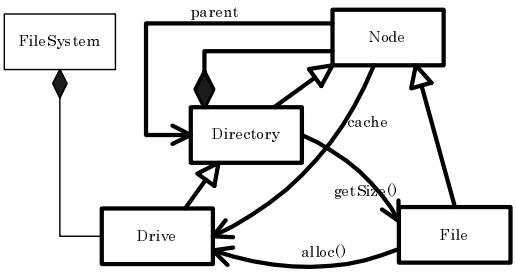
ソースコード縮退の効果として，要素間に依存関係の循環があった場合には，その一部の要素を擬似定数要素に指定することにより，その循環を断ち切ることができる．また，副次的な効果として，ソースコード中の擬似定数要素に関連する部分 (あるいはその部分による影響) 取り除くことで，ソースコードのサイズが減少する．

2.2 例題プログラム

循環する依存関係が存在するプログラムを例題に用いて，ソースコード縮退によってソースコードがどのように変更されるかを説明する．

Table 1: 例題プログラムのクラス

クラス	説明
Directory	ディレクトリ．いくつかの子ノードを持つ．
Drive	ディスクドライブ．ルートディレクトリであり， ファイルを格納する領域を割り当てる機能も持つ．
File	ファイル．
FileSystem	ファイルシステム．いくつかのドライブを持つ．
Node	ノードは，ファイルとディレクトリに共通する機能 を持つ抽象クラス．一つの親ディレクトリを持つ．



太線はクラス間の依存関係が循環している部分

Figure 1: 例題プログラムのクラス間の依存関係

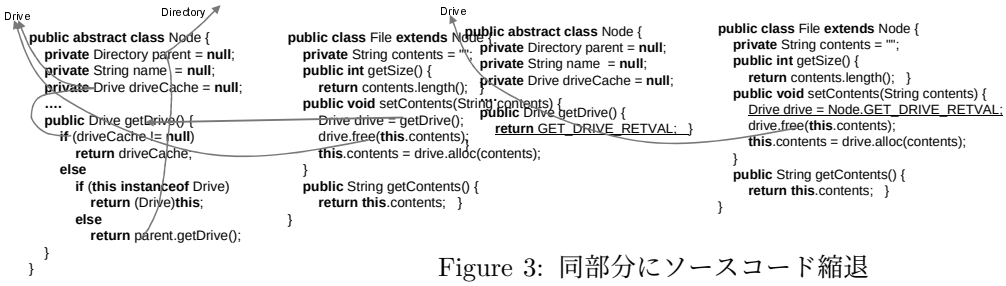


Figure 2: 例題プログラムのソースコードの一部

Figure 3: 同部分にソースコード縮退を適用したもの

対象となるソースコードは，ファイルシステムを模倣するプログラムであり，Java 1.4 で記述され，全体で 380 行程度である (A 参照)．ファイルシステムの機能として，階層的なディレクトリを作成すること，ファイルを作成すること，ファイルを別のディレクトリに移動すること，ファイルやディレクトリの名前を変更すること，ドライブの残り容量を求めることができる．

例題プログラムに含まれる全クラスとその機能を表 1 に示す．クラス間の依存関係を UML のクラス図上で表したものを図 1 に示す．これらのクラス間に生じる依存関係としては，まず，階層的なディレクトリ構造（木構造）を表現するための，`Node` と`Directory` の間の依存関係があげられる．それ以外の依存関係の例としては，`Node` は自身を保持している`Drive` を記憶するためにインスタンス変数`driveCache` を持ち，`Directory` は`File` のサイズを問い合わせるためのメソッド`File.getSize()` を呼び出し，`File` は記憶領域を確保するため，メソッド `Drive.alloc()` を呼び出す．結果として，`Directory`，`Drive`，`File`，`Node` の 4 つのクラスの間に循環する依存関係が存在することになる．したがって，たとえばスライスや Concept Lattice として`File` クラスとそれに関連する部分を抜き出そうとすると，これら 4 つのクラスが取り出されることになる．

図 2 に例題プログラムのソースコードを示す．図では，クラス`Node` の一部と `File` の全体が含まれている．図中の矢印で示されるように，クラス File のソースコードには，`Node` のメソッドの呼び出しが含まれ，さらに，`Node` はインスタンス変数`driveCache` を通じて Drive を参照する，という依存関係が含まれている．ここで，クラス`Node` を擬似定数要素に指定してソースコード縮退を行うと，この部分は図 3 のように書き換えられる．`File` のソースコードに含まれていたクラス`Node` のメソッド呼び出しは，定数`Node.GET_DRIVE_RETVAL` の参照に置き換えられ，`File` から`Node` への依存関係が消える．

2.3 ソースコード縮退の応用

ソースコード縮退はソフトウェアを構成する要素間の循環する依存関係を断ち切る技術であり，その応用として，インクリメンタルなコードレビューを提案する．これには，(1) まず，他に依存しない少数の要素を抽出し，それらの要素をレビュー

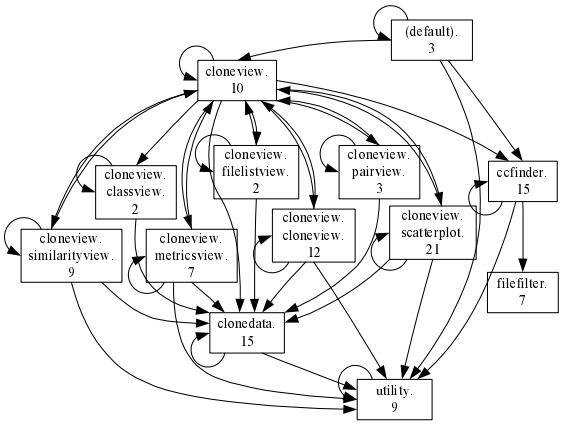


Figure 4: Gemini を構成するパッケージ間の依存関係と個々のパッケージに含まれるクラスの数

によって理解する．(2) 次に，すでにレビューした要素のみに依存する要素を抽出し，レビューにより理解する．これを，すべての要素をレビューし尽くすまで繰り返す．次章ではケーススタディによりこのプロセスを具体的に説明する．

ソースコード縮退を，依存関係に基づいて要素の一部を取り出すプログラムスライシングや Concept Lattice といった手法に応用すれば，これらの手法において循環する依存関係のゆえにまとめて取り出されていたような構成要素群を分割し，より少数の要素を取り出すことが可能である．

3　ケーススタディ

提案する手法を，大阪大学井上研究室で開発されたアプリケーション Gemini[10] に適用する．Gemini は GUI を備えたソースコード分析ツールである．規模は Java ソースコード約 1 万 5 千行，クラス 115 個，パッケージ 13 個である．Gemini のソースコードを理解するために，ソースコード縮退を用いながら，これらのクラスを順にレビューするプロセスを示す．

図 2 に例題プログラムのソースコードを示す．図では，クラス`Node` の一部と `File` の全体が含まれている．図中の矢印は何らかの参照を示している．例えば，クラス`File` のソースコードには，`Node` のメソッドの呼び出しが含まれ，さらに，`Node` はインスタンス変数`driveCache` を通じて`Drive` を参照する，という依存関係が含まれている．ここで，クラス`Node` を擬似定数要素に指定してソースコード縮退を行うと，この部分は図 3 のように書き換えられる．`File` のソースコードに含まれていたクラス`Node` のメソッド呼び出しは，定数`Node.GET_DRIVE_RETVAL` の参照に置き換えられ，`File` から`Node` への依存関係が消える．

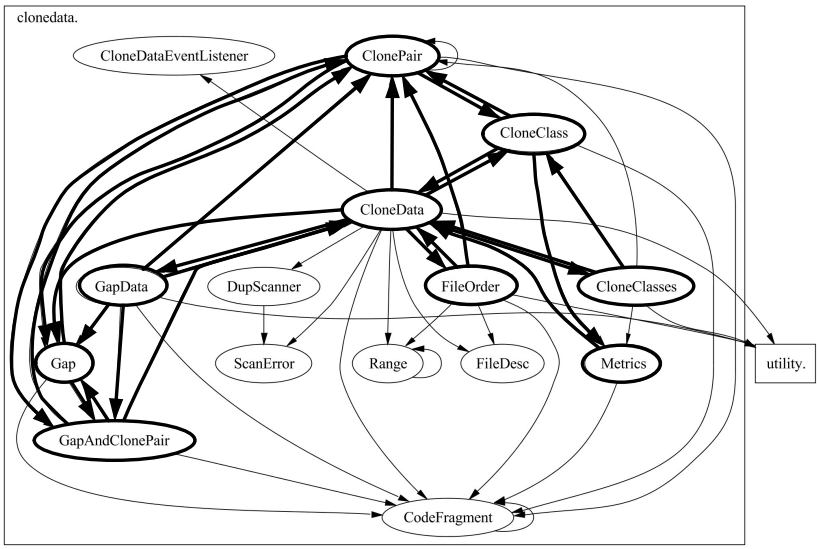


Figure 5: パッケージ clonedata に含まれるクラス間の依存関係

次の段階として，パッケージ`clonedata`をレビューするために，同パッケージに含まれる 15 個のクラス (およびインターフェイス) の依存関係を調べる (図 5)．半数以上のクラスが依存関係の循環 (例えば，`ClonePair` → `GapAndClonePair` → `CloneData` → `ClonePair`) を起こしていることがわかる．これら依存関係の循環を起こしているクラス群{ `CloneClass`, `CloneClasses`, `CloneData`, `ClonePair`, `FileOrder`, `Gap`, `GapAndClonePair`, `GapData`, `Metrics` }の中でもっとも多くの依存を保持しているクラス`CloneData`を擬定数要素に指定してソースコード縮退を行うことにした．適用結果を図 6 に示す．依存関係の循環を起こすクラスは{ `CloneClass`, `ClonePair`, `Gap`, `GapAndClonePair` }に減少している．

行数による評価として，ソースコード縮退前後のソースファイルの行数を表 2 に示す．ソースコード縮退によって，依存関係が循環を引き起こしているクラス群の総行数は，3265 行から 542 行へと減少していることがわかる．また，行数が最大のソースファイル`FileOrder.java`は，約 4 割の行が取り除かれることがわかる．

レビューの手順としては，依存関係の循環を起こしているクラス群の 4 つのクラスを同時にレビューする必要があることを除けば，疑似定数要素`CloneData`以外の 14 個のクラスのソースコード縮退後のソースコードを，依存関係に基づいて一つずつ順にレビューすることになる．これら 14 個のクラスのレビューが終わった時点で，ソースコード縮退によって取り除かれたていた部分をレビューする．これによりパッケージ`clonedata`のレビューが完了する．

ソースコード縮退を適用する場合と，適用しない場合とで，レビューがどの

Table 2: ソースコード縮退前後のソースファイルの行数

ソースファイル	LOC	縮退後の LOC
CloneClass.java	*109	*109
CloneClasses.java	*465	462
CloneData.java	*596	0
CloneDataEventListener.java	6	6
ClonePair.java	*129	*129
CodeFragment.java	79	79
DupScanner.java	195	195
FileDesc.java	19	19
FileOrder.java	*1217	741
Gap.java	*104	*104
GapAndClone.java	*202	*200
GapData.java	*228	127
Metrics.java	*215	208
Range.java	25	25
ScanError.java	7	7
総 LOC	3596	2411
依存関係の循環に含まれるクラスの総 LOC	*3265	*542

「*」がつけられた数字はそのクラスが循環する依存関係に含まれることを意味する．

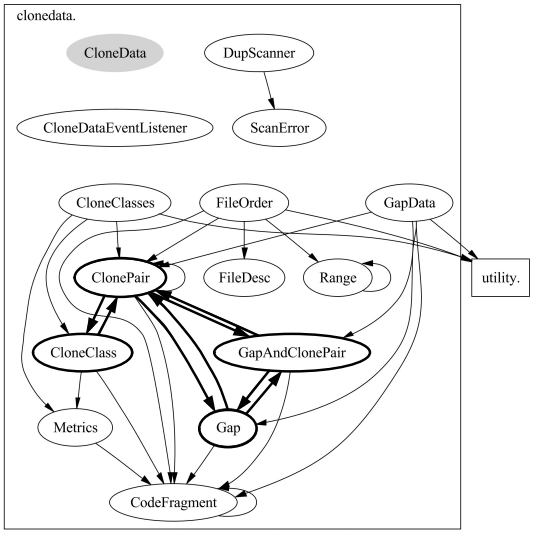


Figure 6: ソースコード縮退適用後の依存関係

ように進むかを比較したものを図 7 に示す．依存関係の循環を起こしているクラスを一度にレビューすると，ソースコード縮退を行わない場合は，最後に 9 つのクラスを一度にレビューすることになり，険しい学習曲線となることがわかる．

残ったパッケージに関しても，同様の手順を行うことでレビューを行うことができる．

4 関連研究

文献 [4] には，プログラミング言語 C++で大規模なソフトウェアを開発する場合に，依存関係の循環ななぜ起こるか，なぜ避けるべきなのか，どのような場合にどうすれば避けることができるのかを例を挙げて説明している．

文献 [7] には，オブジェクト指向プログラミング言語（C++や Java）のリバースエンジニアリングにおいて，実装（ソースコード）におけるクラスの記述と，設計におけるクラスの記述の抽象度が異なることによって生じる問題が指摘されている．また，この問題を解決するために，リバースエンジニアリングによって得られる情報に抽象的な情報を付加することで，より設計に近いクラス図を復元するための手法を提案している．

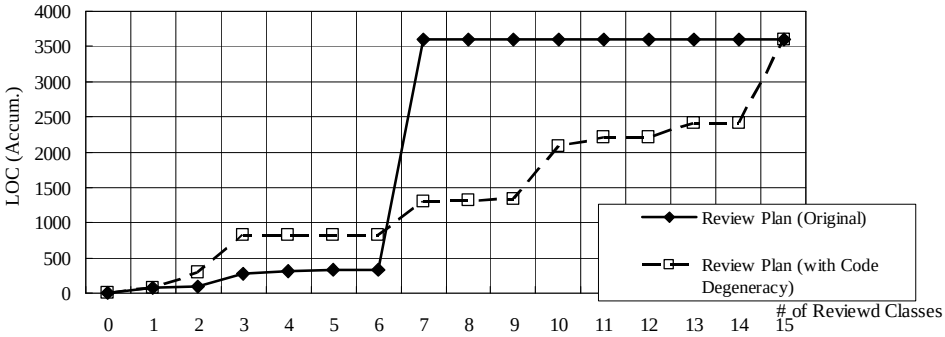


Figure 7: 学習曲線の比較

5 まとめ

本稿では，ソースコードを理解するための分析手法の一つである「ソースコード縮退」を提案した．本手法により，他の要素への依存関係を持たない要素から順に，一度に少数の要素をレビューしていくことで，インクリメンタルにソースコード全体をレビューすることが可能になる．ケーススタディでは，ある Java プログラムに本手法を適用することによって，そのようなレビューの具体例を示した．

今後の展開としては，本手法を統合開発環境に組み込むことでより手軽に適用できるようにすること，他の依存関係に基づく分析手法と組み合わること，あるいは，ソフトウェア分析のための枠組みである Reflextion model[5] 等において本手法を利用することがある．

また，アスペクト指向技術によって開発されたプログラムでは，アスペクトを導入することにより，クラスとアスペクトの間に新たな依存関係の循環が生じる事例が知られており [3]，そのようなソフトウェアをレビューする際にも本手法を用いることが可能であろう．

ソースコード縮退を分離テスト [4] に応用することも可能であろう．すなわち，元のプログラムの開発者が，テストの一環として，適切な擬似定数要素を選択してソースコード縮退を行い，その入出力とともにテストケースとして残しておく．ソースコード縮退を行うことにより，特定の入力に対する処理がソースコードのどの部分で行われるべきかを，ドキュメントではなく（縮退された）ソースコードとして記述することになる．このような，ソースコードとして残されるホワイトテストは，ソースコードにコメントとして埋め込まれるドキュメントと同程度に，ソースコードへの追従が正確であることが期待される．

長期的な課題としては，ソースコード縮退（および一般には，ソースコードの一部を取り出す技術）の大規模ソフトウェアを対象とした場合の有効性評価がある．これらの技術を用いて一部を取り出したり修正することにより，ソースコードの理解が妨げられないかを評価する必要がある．元のソースコードと，取り出されたり修正されたコードとの対応を，ユーザーインターフェイスとしてどのよ

うに見せるかも重要な問題である．

References

[1] Eisenbarth, T., Koschke, R., Simon, D.: Locating Features in Source Code, *IEEE Trans. Software Engineering*, vol. 29, no. 3, pp. 210-224 (2003).

[2] 井上克郎, 神谷年洋, 楠本真二: コードクローン検出法, コンピュータソフトウェア, Vol.18, No.5, pp.47-54, (2001).

[3] 石尾隆, 楠本真二, 井上克郎: アスペクト指向プログラミングのプログラムスライス計算への応用, 情報処理学会 論文誌 Vol. 44, No. 7, pp. 1709-1719 (2003-7).

[4] Lakos, J.: *Large-Scale C++ Software Design*, Addison-Wesley (1999).
ジョン ラコス (著), 滝沢 徹, 牧野 祐子 (翻訳): 大規模 C++ソフトウェアデザイン, ピアソンエデュケーション (1999).

[5] Murphy, G. C., Notkin, D., Sullivan, K.: Software Reflexion Models: Bridging the Gap between Source and High-Level Models, *ACM SIGSOFT '95: Proc. the 3rd Symposium on the Foundations of Software Engineering (FSE)*, pp. 18-28, Washington D. C. (Oct. 1995).

[6] 大畑文明, 近藤和弘, 井上克郎: エイリアスフローグラフを用いたオブジェクト指向プログラムのエイリアス解析手法, 電子情報通信学会論文誌 D-I, Vol. J84-D-I, No.5, pp. 443-453 (2001).

[7] 齊藤寿和, 海尻賢二: オブジェクト指向プログラムのリバースエンジニアリング, 研究報告 ソフトウェア工学, 136-016, pp. 119-126 (2002).

[8] 高田智規, 井上克郎, 芦田佳行, 大畑文明: 制限された動的情報を用いたプログラムスライシング手法の提案，電子情報通信学会論文誌 D-I, Vol. J85-D-I, No. 2, pp. 228-235, (2002).

[9] 竹村司: UML クラス図からのワークロードの見積もりとスケジュール作成, オブジェクト指向 2003 シンポジウム予稿集, pp.153-160 (2003).

[10] Ueda, Y., Higo, Y., Kamiya, T., Kusumoto, S., and Inoue, K.: Gemini: Code Clone Analysis Tool, *Proc. 2002 International Symposium on Empirical Software Engineering (ISESE2002)*, vol.2, pp.31-32, Nara, Japan, (Oct. 2002).

[11] Weiser, M.: Program Slicing, *Proc. the Fifth International Conference on Software Engineering*, pp. 439-449, (1981).

[12] 山本哲男, 横森励士, 松下誠, 楠本真二, 井上克郎: 利用頻度に基づくソフトウェア部品の解析・検索システムの提案, 電子情報通信学会技術研究報告, SS2002-17, Vol.102, No.329, pp.13-18 (2002).

A　例題プログラムのソースファイル

2.2 で用いた例題プログラムのソースコード全体を示す．紙面の都合により，一部のコメント，空白文字（スペース，タブ，改行），{}を取り除いた．asssert 文を用いているため，コンパイルには javac 1.4 以降が必要である．

A.1　filesystem/Directory.java

```
package filesystem;
import java.util.*;
public class Directory extends Node {
  private HashMap entities = new HashMap();
  public String[] getEntityNames() {
    return (String[])entities.keySet().toArray(new String[0]);
  }
  public Node getEntitiy(String name) {
    assert entities.containsKey(name);
    return (Node)entities.get(name);
  }
  public File createFile(String name) {
    assert ! entities.containsKey(name);
    File file = new File();
    entities.put(name, file);
    file.setParent(this);   file.setName(name);
    return file;
  }
  public Directory createSubDirectory(String name) {
    assert ! entities.containsKey(name);
    Directory subdir = new Directory();
    entities.put(name, subdir);
    subdir.setParent(this);   subdir.setName(name);
    return subdir;
  }
  public int getContentsSize() {
    int size = 0;
    Set keys = entities.keySet();
    Iterator i = keys.iterator();
    while (i.hasNext()) {
      Node n = (Node)entities.get(i.next());
      if (n instanceof Directory)
        size += ((Directory)n).getContentsSize();
      else if (n instanceof File)
        size += ((File)n).getSize();
      else assert false;
    }
    return size;
  }
  public void delete(String name) {
    assert entities.containsKey(name);
    Node n = (Node)entities.get(name);
    if (n instanceof Directory)
      ((Directory)n).delete_i();
    else if (n instanceof File)
```

```
      ((File)n).setContents("");
    else   assert false;
    entities.remove(name);
  }
  private void delete_i() {
    Set keys = entities.keySet();
    Iterator i = keys.iterator();
    while (i.hasNext()) {
      Node n = (Node)entities.get(i.next());
      if (n instanceof Directory) ((Directory)n).delete_i();
      else if (n instanceof File) ((File)n).setContents("");
      else assert false;
    }
    entities.clear();
  }
  Node remove(String name) {
    assert entities.containsKey(name);
    return (Node)entities.remove(name);
  }
  void put(String name, Node n) {
    assert ! entities.containsKey(name);
    assert n.getDrive().equals(getDrive());
    entities.put(name, n);
} }
```

A.2 filesystem/Drive.java

```
package filesystem;
public class Drive extends Directory {
  private int capacity = 100;
  private int used = 0;
  public void setParent(Node node) {
    assert false; // a drive does not have a parent dir.
  }
  public String getPath() { return getName(); }
  public String alloc(String contents) {
    used += contents.length();
    return contents;
  }
  public void free(String contents) { used -= contents.length(); }
  public int getCapacity() { return capacity; }
  public int getUsed() { return used; }
  public int getRemain() { return capacity - used; }
}
```

A.3 filesystem/File.java

```
package filesystem;
public class File extends Node {
  private String contents = "";
  public int getSize() { return contents.length(); }
  public void setContents(String contents) {
    Drive drive = getDrive();
    drive.free(this.contents);
```

```
    this.contents = drive.alloc(contents);
  }
  public String getContents() { return contents; }
}
```

A.4 filesystem/FileSystem.java

```
package filesystem;
import java.util.HashMap;
public class FileSystem {
  private HashMap drives = new HashMap();
  public String[] getDriveNames() {
    return (String[])drives.keySet().toArray(new String[0]);
  }
  public void putDrive(String name, Node node) {
    drives.put(name, node);
  }
  public Drive getDrive(String name) {
    return (Drive)drives.get(name);
  }
  public Drive createDrive(String driveName) {
    assert ! drives.containsKey(driveName);
    Drive drive = new Drive();
    drive.setName(driveName);
    return drive;
  }
  public static void move(Directory srcDir, String name,
      Directory destDir) {
    Node n = srcDir.getEntitiy(name);
    if (n.getDrive().equals(destDir.getDrive())) {
      // error if srcDir is one of the sub-dir of destDir.
      // error if destDir is one of the sub-dir of srcDir.
      Node detachedN = srcDir.remove(name);
      destDir.put(name, detachedN);
    } else {
      FileSystem.copy(srcDir, name, destDir);
      srcDir.delete(name);
  } }
  public static void copy(Directory srcDir, String name,
      Directory destDir) {
    Node n = srcDir.getEntitiy(name);
    if (n instanceof File) {
      File srcF = (File)n;
      File destF = destDir.createFile(name);
      destF.setContents(srcF.getContents());
    } else if (n instanceof Directory) {
      Directory srcD = (Directory)n;
      Directory destD = destDir.createSubDirectory(name);
      String[] entities = srcD.getEntityNames();
      for (int i = 0; i < entities.length; ++i)
        FileSystem.copy(srcD, entities[i], destD);
} } }
```

A.5 filesystem/Node.java

```
public abstract class Node {
  private Directory parent = null;
  private String name  = null;
  private Drive driveCache = null;
  public void setParent(Directory parent) {
    if (this.parent != null)
      assert this.parent.getDrive().equals(parent.getDrive());
    this.parent = parent;
    driveCache = parent.getDrive();
  }
  public Directory getParent() { return parent; }
  public void setName(String name) { this.name = name; }
  public String getName() { return name; }
  public String getPath() {
    String path = parent.getPath();
    return path + "/" + name;
  }
  public Drive getDrive() {
    if (driveCache != null) return driveCache;
    else
      if (this instanceof Drive) return (Drive)this;
      else return parent.getDrive();
} }
```

A.6 Test1.java

```
import filesystem.*;
public class Test1 {
  public static void main(String[] args) {
    String TRBOD = "The remaining block of drive '";
    java.io.PrintStream out = System.out;
    FileSystem fs = new FileSystem();
    Drive driveA = fs.createDrive("a:");
    out.println("A drive " + driveA.getName()
        + " has been created.");
    Directory dirTemp = driveA.createSubDirectory("temp");
    out.println("A directory " + dirTemp.getPath()
        + " has been created.");
    File fileB = dirTemp.createFile("b.txt");
    fileB.setContents("12345");
    out.println("A file " + fileB.getPath()
        + " has been created.");
    File fileC = dirTemp.createFile("c.txt");
    fileC.setContents("abc");
    out.println("An another file " + fileC.getPath()
        + " has been created.");
    String path = dirTemp.getPath();
    int size = dirTemp.getContentsSize();
    out.println("The size of directory '" + path + "' is "
        + size);
    int remain = driveA.getRemain();
    out.println(TRBOD + driveA.getName() + "' is " + remain);
    Drive driveD = fs.createDrive("d:");
```

```
    out.println("A drive " + driveD.getName()
        + " has been created.");
    FileSystem.move(dirTemp, "b.txt", driveD);
    out.println("A file has moved from drive a: to d:");
    int remainA = driveA.getRemain();
    out.println(TRBOD + driveA.getName() + "' is " + remainA);
    int remainD = driveD.getRemain();
    out.println(TRBOD + driveD.getName() + "' is " + remainD);
    System.exit(0);
} }
```