Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Aufgabe 3 (Überschreiben, Überladen und Verdecken): (6 + 5 = 11 Punkte)

Betrachten Sie die folgenden Klassen:

Listing 1: A.java

```
1  public class A {
2      public final String x;
3
4      public A() {                                    // Signatur: A()
5          this("written in A()");
6      }
7
8      public A(int p1) {                              // Signatur: A(int)
9          this("written in A(int)");
10     }
11
12     public A(String x) {                            // Signatur: A(String)
13         this.x = x;
14     }
15
16     public void f(A p1) {                           // Signatur: A.f(A)
17         System.out.println("called A.f(A)");
18     }
19 }
```

Listing 2: B.java

```
1  public class B extends A {
2      public final String x;
3
4      public B() {                                    // Signatur: B()
5          this("written in B()");
6      }
7
8      public B(int p1) {                              // Signatur: B(int)
9          this("written in B(int)");
10     }
11
12     public B(A p1) {                                // Signatur: B(A)
13         this("written in B(A)");
14     }
15
16     public B(B p1) {                                // Signatur: B(B)
17         this("written in B(B)");
18     }
19
20     public B(String x) {                            // Signatur: B(String)
21         super("written in B(String)");
22         this.x = x;
23     }
24
25     public void f(A p1) {                           // Signatur: B.f(A)
26         System.out.println("called B.f(A)");
27     }
28
29     public void f(B p1) {                           // Signatur: B.f(B)
30         System.out.println("called B.f(B)");
31     }
32 }
```

Listing 3: C.java

```
1  public class C {
2      public static void main(String[] args) {
3                                                      // a)
4          A v1 = new A(100);                          // (1)
5          System.out.println("v1.x: " + v1.x);
6
7          A v2 = new B(100);                          // (2)
8          System.out.println("v2.x: " + v2.x);
9          System.out.println("((B) v2).x: " + ((B) v2).x);
10
11         B v3 = new B(v2);                           // (3)
12         System.out.println("((A) v3).x: " + ((A) v3).x);
```

```
13          System.out.println("v3.x: " + v3.x);
14
15          B v4 = new B();                                     // (4)
16          System.out.println("((A) v4).x: " + ((A) v4).x);
17          System.out.println("v4.x: " + v4.x);
18
19                                                              // b)
20          v1.f(v1);                                           // (1)
21          v1.f(v2);                                           // (2)
22          v1.f(v3);                                           // (3)
23          v2.f(v1);                                           // (4)
24          v2.f(v2);                                           // (5)
25          v2.f(v3);                                           // (6)
26          v3.f(v1);                                           // (7)
27          v3.f(v2);                                           // (8)
28          v3.f(v3);                                           // (9)
29      }
30  }
```

(7)
v3 deklariert als B und in Laufzeit als B; v1 deklariert als A
=> passende Methoden A.f(A), B.f(A)
=> B.f(A)

(8)
v3 deklariert als B und in Laufzeit als B; v2 deklariert als A
A.f(A), B.f(A)
=> B.f(A)

(9)
v3 deklariert als B und in Laufzeit als B; v3 deklariert als B
A.f(A), B.f(A), B.f(B)
=> B.f(A)

In dieser Aufgabe sollen Sie angeben, welche Methoden- und Konstruktoraufrufe stattfinden, wenn die `main`-Methode der Klasse `C` ausgeführt wird. Benutzen Sie hierzu keinen Computer, sondern nur die aus der Vorlesung bekannten Angaben zum Verhalten von Java. Verwenden Sie zur eindeutigen Bezeichnung die Funktionssignatur, die jeweils als Kommentar hinter jeder Funktionsdefinition steht. Begründen Sie Ihre Antwort kurz.

a) Geben Sie für die mit `(1)-(4)` markierten Konstruktoraufrufe in der Klasse `C` jeweils an, welche Konstruktoren in welcher Reihenfolge von Java aufgerufen werden. Notieren Sie auch die von Java implizit aufgerufenen Konstruktoren. Bedenken Sie, dass die Oberklasse von `A` die Klasse `Object` ist.

b) Geben Sie für die mit `(1)-(9)` markierten Aufrufe der Methode `f` in der Klasse `C` jeweils an, welche Variante der Funktion von Java verwendet wird. Geben Sie hierzu die jeweilige Signatur an.

Lösung:

a) (1) Ausgeführte Konstruktoren:

`A(int)` explizit
`A(String)` explizit
`Object()` implizit

Ausgabe:

```
v1.x: written in A(int)
```

(2) Ausgeführte Konstruktoren:

`B(int)` explizit
`B(String)` explizit
`A(String)` explizit
`Object()` implizit

Ausgabe:

```
v2.x: written in B(String)
((B) v2).x: written in B(int)
```

(3) Ausgeführte Konstruktoren:

`B(A)` explizit
`B(String)` explizit
`A(String)` explizit
`Object()` implizit

Ausgabe:

```
((A) v3).x: written in B(String)
v3.x: written in B(A)
```

(4) Ausgeführte Konstruktoren:

`B()` explizit

`B(String)` explizit

`A(String)` explizit

`Object()` implizit

Ausgabe:

```
((A) v4).x: written in B(String)
v4.x: written in B()
```

b) (1) `v1.f(v1);`
Typen:

- `v1`: deklarierter Typ `A`, Laufzeittyp `A`
- `v1`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`
Aufgerufene Methode: `A.f(A)`

(2) `v1.f(v2);`
Typen:

- `v1`: deklarierter Typ `A`, Laufzeittyp `A`
- `v2`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`
Aufgerufene Methode: `A.f(A)`

(3) `v1.f(v3);`
Typen:

- `v1`: deklarierter Typ `A`, Laufzeittyp `A`
- `v3`: deklarierter Typ `B`

Passende Methoden: `A.f(A)`
Aufgerufene Methode: `A.f(A)`

(4) `v2.f(v1);`
Typen:

- `v2`: deklarierter Typ `A`, Laufzeittyp `B`
- `v1`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(5) `v2.f(v2);`
Typen:

- `v2`: deklarierter Typ `A`, Laufzeittyp `B`
- `v2`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(6) `v2.f(v3);`
Typen:

- `v2`: deklarierter Typ `A`, Laufzeittyp `B`
- `v3`: deklarierter Typ `B`

Passende Methoden: `A.f(A)`, `B.f(A)`, `B.f(B)`
Aufgerufene Methode: `B.f(A)`

Da `v2` als `A` deklariert ist, wird die Methode `A.f(A)` ausgewählt. Diese kann vom Laufzeittyp `B` von `v2` nur durch die Methode `B.f(A)` überschrieben werden, nicht aber durch `B.f(B)`. Somit wird, da `A.f(B)` nicht existiert, in der Tat `B.f(A)` aufgerufen, obwohl `v3` als `B` deklariert ist.

(7) `v3.f(v1);`
Typen:

- `v3`: deklarierter Typ `B`, Laufzeittyp `B`
- `v1`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(8) `v3.f(v2);`
Typen:

- `v3`: deklarierter Typ `B`, Laufzeittyp `B`
- `v2`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(9) `v3.f(v3);`
Typen:

- `v3`: deklarierter Typ `B`, Laufzeittyp `B`
- `v3`: deklarierter Typ `B`

Passende Methoden: `A.f(A)`, `B.f(A)`, `B.f(B)`
Aufgerufene Methode: `B.f(B)`

Aufgabe 5 (Klassenhierarchie): (13 Punkte)

Eine der grundlegenden Funktionalitäten des Betriebssystems ist es, einen einfachen und einheitlichen Zugriff auf gespeicherte Daten zu liefern. Dabei muss es der Benutzer*in Ordner (Directories) und Dateien (Files) präsentieren. Im Folgenden sehen Sie ein Beispiel für einen Teil eines typischen Linux Dateisystems.

```
/
/boot/
/boot/kernel
/etc/
/etc/motd
```

Wir sehen den Wurzelordner `/`, die beiden Unterordner `boot` und `etc` sowie die beiden Dateien `kernel` und `motd`[1].
Intern wird der Inhalt einer Datei oder eines Unterordners nicht unter ihrem Namen abgelegt, sondern unter einem sogenannten inode, einem Integer. Der Eintrag im Ordner enthält dann nur die Information, unter welchem inode der Inhalt zu finden ist. Falls beispielsweise der Inhalt der Datei `motd` unter dem inode 5 abgelegt ist, dann steht im Unterordner nur, dass hier eine Datei mit dem Namen `motd` existiert und dass deren Inhalt unter dem inode 5 zu finden ist.
Dies ist ein nützlicher Mechanismus, denn er erlaubt es, auf einfache Art und Weise den Inhalt zweier Dateien gleich zu halten. Dazu wird ein zweiter Eintrag im Ordner angelegt, welcher auf denselben inode verweist wie ein bereits existierender Eintrag. So ist es möglich, einen zweiten Eintrag `friendly-message.txt` im Ordner `etc` zu erstellen, welcher ebenfalls auf den inode 5 verweist. Wird nun der Inhalt von `motd` verändert, so ändert sich automatisch auch der Inhalt von `friendly-message.txt`, und umgekehrt. Man sagt, `friendly-message.txt`

[1] message of the day

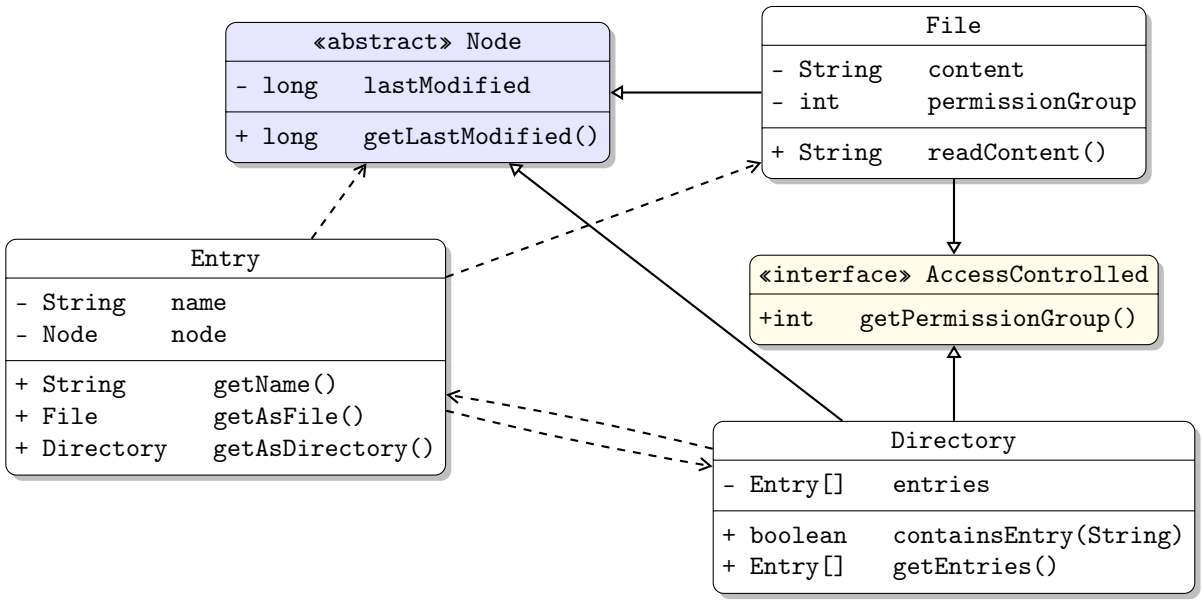
ist ein *Hardlink* auf den Inhalt von `motd`. Auch `motd` ist ein Hardlink auf seinen Inhalt. In der Tat sind alle Einträge im Ordner gleichberechtigte Hardlinks auf ihren Inhalt. Ein inode wird erst dann gelöscht, wenn der letzte auf ihn verweisende Hardlink gelöscht wurde.
Im Folgenden werden wir von Abstraktion sprechen, und damit eine (evtl. abstrakte) Klasse oder ein Interface meinen. Wählen Sie die jeweils geeignetste Variante.
Modellieren Sie ein Dateisystem wie folgt:

- Jeder Hardlink, also jeder Eintrag im Ordner, wird durch eine Abstraktion `Entry` dargestellt. Jedes `Entry`-Objekt hat einen `name` sowie eine Referenz auf einen `Node`.
- Jeder inode, also jeder Inhalt, wird hier nicht durch einen Integer, sondern durch ein Objekt der Abstraktion `Node` dargestellt. Jedes `Node`-Objekt hat das Attribut `lastModified`, welches den Zeitpunkt der letzten Änderung enthält und über die Methode `long getLastModified()` abgerufen werden kann.
- Ein Dateiinhalt ist ein `Node`, welcher durch ein Objekt der Abstraktion `File` dargestellt wird. Jedes `File`-Objekt hält seinen Inhalt in einem `String content`, welcher über die Methoden `String readContent()` gelesen werden kann und ein `int`-Attribut `permissionGroup`, zu dem später mehr erklärt wird.
- Ein Ordnerinhalt ist ein `Node`, welcher durch ein Objekt der Abstraktion `Directory` dargestellt wird. Jedes `Directory`-Objekt hält seine Dateien und Unterordner in einem Array von `Entry`s, welches über die Methode `Entry[] getEntries()` abgerufen werden kann. Über die Methode `boolean containsEntry(String name)` kann geprüft werden, ob der Ordner einen gegebenen Eintrag enthält.
- Die Abstraktion `Entry` bietet ebenfalls Methoden. Die Methode `String getName()` gibt das `name`-Attribut zurück. Die Methode `File getAsFile()` liefert ihren `Node` als `File`. Die Methode `Directory getAsDirectory()` arbeitet analog dazu.
- Sowohl `File` als auch `Directory` sollen Informationen über die Zugriffserlaubnis bieten. Daher sollen beide eine Methode `int getPermissionGroup()` bereitstellen. Bei `File` ergibt sich dies aus der `permissionGroup`, bei `Directory` wird die zugriffsberechtigte Gruppe aus den Ordnerinhalten berechnet. Ergänzen Sie hier ggf. eine passende Abstraktion.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie. Notieren Sie keine Konstruktoren. Die in der Aufgabenstellung erwähnten Getter und Setter sollen notiert werden, aber andere nicht. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden.
Verwenden Sie hierbei die gleiche Notation bzw. Darstellungsweise wie in Tutoraufgabe 4.

Lösung:

Die Zusammenhänge können wie folgt modelliert werden:



Aufgabe 7 (Programmieren mit Klassenhierarchien): (1 + 2 + 6 + 7 + 4 + 2 + 4 = 26 Punkte)

Ziel dieser Aufgabe ist es, eine einfache Versionsverwaltung[2] zu implementieren. Diese verwaltet beliebig viele Versionen beliebig vieler Text-Dateien, die in demselben Verzeichnis liegen. Dieses bezeichnen wir im Folgenden als Wurzelverzeichnis. Alle von der Versionsverwaltung erfassten Versionen werden in dem Unterverzeichnis `vcs` des Wurzelverzeichnisses gespeichert. Dieses bezeichnen wir im Folgenden als Backupverzeichnis. Die neueste in die Versionsverwaltung eingecheckte Version ist stets direkt im Backupverzeichnis gespeichert. Wenn eine neue Version eingecheckt wird, wird im Backupverzeichnis ein neues Unterverzeichnis erstellt, in das die letzte eingecheckte Version verschoben wird. Die Versionsverwaltung erlaubt das Ausführen folgender Kommandos:

- `listfiles`: Gibt die Namen aller Dateien im Wurzelverzeichnis (aber nicht im Backupverzeichnis oder weiteren Unterverzeichnissen) aus.
- `commit`: Checkt eine neue Version in die Versionsverwaltung ein, d.h.,
  - es wird ein neues Unterverzeichnis im Backupverzeichnis erstellt,
  - alle Dateien im Backupverzeichnis werden in das neue Unterverzeichnis verschoben und
  - alle Dateien im Wurzelverzeichnis werden in das Backupverzeichnis kopiert.
- `exit`: Beendet die Anwendung.

Die Interaktion mit der Versionsverwaltung kann also z.B. wir folgt aussehen, wobei `./repository` das Wurzelverzeichnis ist. Die grauen Zahlen am Ende der Zeile gehören dabei nicht zur Aus-/Eingabe, sie dienen als Referenz, um nach der Ausgabe die jeweils resultierende Ordnerstruktur zu zeigen.

```
java Main ./repository/ 1
initialized empty repository
> listfiles 2
test2
test
> commit 3
Committed the following files:
test2
test
>commit 4
Committed the following files:
test2
test
> exit
```

Vor dem Starten des Programms ist `./repository` ein leerer Ordner. Nach 1 existiert ein neuer Unterordner `./repository/vcs`, ansonsten gibt es keine Dateien. Nehmen wir an, dass nach 1 außerhalb des Programms von einer Nutzer*in zwei Dateien `test` und `test2` erstellt und in `./repository/` abgelegt wurden. Es existieren also nur die Dateien `./repository/test1` und `./repository/test2` und diese werden ohne eine Änderung der Ordnerstruktur von `listfiles` nach Ausführen von 2 ausgegeben. Bei 3 werden nun diese beiden Dateien commited, d.h. sie werden in `./repository/vcs` kopiert und es wird ein neuer Unterordner `./repository/vcs/123456789` erstellt, wobei `123456789` stellvertretend für einen Unix-Timestamp steht. Dies wird später näher erläutert. In diesen Ordner werden alle vorigen Dateien aus `./repository/vcs` verschoben, in unserem Beispiel bleibt der Ordner `./repository/vsc/123456789` leer, da zuvor keine Dateien committed wurden. Bei Befehl 4 werden nun erneut zwei, potentiell geänderte, Versionen von `test2` und `test` commited. Es wird nun also ein neuer Unterordner `./repository/vcs/4242424242` erstellt, in den nun die alten Versionen von `test` und `test2` verschoben werden. Die aktuellen Versionen von `test` und `test2` werden wiederum in `./repository/vcs` kopiert.
Zum Lösen der folgenden Aufgaben müssen Sie die Klassen `Util`, `VCS`[3], `Command` und `Main` aus dem Moodle herunterladen. Die Klasse `VCS` repräsentiert eine Versionsverwaltung und stellt die beiden Methoden `getRootDir`

[2] `https://de.wikipedia.org/wiki/Versionsverwaltung`
[3] "Version Control System"

und `getBackupDir` zur Verfügung, die den Pfad zum Wurzelverzeichnis bzw. zum Backupverzeichnis zurückliefern. Die Klasse `Main` enthält die `main`-Methode der Versionsverwaltung. Die Klasse `Command` repräsentiert die oben genannten Kommandos. Die Klasse `Util` stellt einige Hilfsmethoden zur Verfügung, die zum Lösen der folgenden Aufgaben benötigt werden. Beachten Sie beim Lösen der folgenden Aufgaben die Prinzipien der Datenkapselung. Sie dürfen beliebig viele zusätzliche Methoden zur Klasse `Command` und den von Ihnen implementierten Klassen (aber nicht zu den anderen vorgegebenen Klassen) hinzufügen.

a) Ergänzen Sie die Implementierung der abstrakten Klasse `Command` um ein Attribut `VCS vcs`. Dieses soll dem Konstruktor als einziges Argument übergeben werden.

b) Implementieren Sie eine Klasse `Exit`, die von `Command` erbt (d.h., `Exit` ist eine Unterklasse von `Command`). Ihre `execute` Methode soll die Anwendung beenden.

Hinweise:

- Die Methode `exit` der Klasse `Util` ist zum Lösen dieser Aufgabe nützlich.

c) Implementieren Sie eine Klasse `ListFiles`, die von `Command` erbt. Die `execute` Methode dieser Klasse soll die Namen aller Dateien im Wurzelverzeichnis der Versionsverwaltung `vcs` ausgeben.

Hinweise:

- Die Methode `listFiles` der Klasse `Util` ist zum Lösen dieser Aufgabe hilfreich.

d) Implementieren Sie eine Klasse `Commit`, die von `ListFiles` erbt. Ihre `execute` Methode soll gemäß der Beschreibung des Kommandos "`commit`" eine neue Version in die Versionsverwaltung einchecken. Anschließend soll sie die Meldung "`Committed the following files:`" gefolgt von einer Liste aller Dateien, die aus dem Wurzelverzeichnis in das Backupverzeichnis kopiert wurden, ausgeben. Verwenden Sie hierzu die Funktionalität, die bereits in `ListFiles.execute` implementiert wurde, wieder. Der Name des neuen Unterverzeichnisses des Backupverzeichnisses soll der aktuelle Unix-Timestamp[4] sein. Diesen liefert die Methode `getTimestamp` der Klasse `Util`.

Hinweise:

- Die Methoden `appendFileOrDirName, mkdir, listFiles, copyFile` und `moveFile` der Klasse `Util` sind zum Lösen dieser Aufgabe nützlich.

e) Implementieren Sie die Methode `Command parse(String cmdName, VCS vcs)` in der Klasse `Command`. Diese soll eine geeignete Instanz der Klasse `Exit` zurückgeben, falls `cmdName` der String "`exit`" ist, sie soll eine geeignete Instanz der Klasse `ListFiles` zurückgeben, falls `cmdName` der String "`listfiles`" ist und sie soll eine geeignete Instanz der Klasse `Commit` zurückgeben, falls `cmdName` der String "`commit`" ist. Andernfalls soll sie eine geeignete Fehlermeldung ausgeben und `null` zurückgeben.

f) Da in Zukunft jede neue `Command`-Art den Status der Dateien nach Ausführung angeben soll und das auf eindeutige Weise, wollen Sie lediglich Vererbung vom Typ `ListFiles` erlauben (d.h. `ListFiles` darf beliebige Unterklassen haben). `Exit` und `Command` sollen, bis auf den bisherigen Stand, nicht weiter erbbar sein (d.h., sie sollen in Zukunft keine weiteren Unterklassen mehr bekommen können). Ändern Sie die vorgegebene Klassendefinitionen von `Command` und passen Sie auch die von Ihnen geschriebenen Klassen `Exit` und `ListFiles` an.

g) Erstellen Sie eine Abstraktion `Modifying`, die von `Commit` implementiert werden soll. Diese soll für alle Kommandos, die das Dateisystem oder Dateien verändern, eine Methodensignatur `String getInformation()` vorgeben. Diese Methode soll bei den entsprechenden Kommandos (in dieser einfachen Version einer Versionsverwaltung ist dies nur `Commit`) einen `String` zurückgeben, der angibt, welche Art von Dateioperationen durchgeführt werden. Ergänzen Sie an der angegebenen Stelle in der `main`-Methode Anweisungen, die dafür sorgen, dass immer wenn ein `Command` ausgeführt werden soll, der diese Methode bereitstellt, diese Informationen vorher ausgegeben werden. Passen Sie außerdem `Commit` so an, dass die neu erstellte Abstraktion genutzt wird. Beispielsweise könnte eine Ausgabe so aussehen:

```
> commit
This command does the following modifying operations:
Files: Copy and Move
```

[4] `https://de.wikipedia.org/wiki/Unixzeit`

```
Directory: create
Committed the following files:
text1.text
text2.text
```

Wenn Sie alle Teilaufgaben gelöst haben, können Sie die Versionsverwaltung ausfüllen, indem Sie `java Main root_directory` ausführen, wobei `root_directory` der Pfad zum Wurzelverzeichnis ist.

Hinweise:

- Da die Versionsverwaltung schreibend auf das Dateisystem zugreift, sollte sie nicht mit einem Wurzelverzeichnis getestet werden, das wichtige Daten enthält.

Lösung:

Listing 4: Command.java

```
1  public sealed abstract class Command permits ListFiles,Exit {
2
3      private VCS vcs;
4
5      public Command(VCS vcs) {
6          this.vcs = vcs;
7      }
8
9      public abstract void execute();
10
11     public VCS getVCS() {
12         return vcs;
13     }
14
15     public static Command parse(String cmdName, VCS vcs) {
16         return switch (cmdName) {
17             case "commit" ->  new  Commit(vcs);
18             case "listfiles" ->  new ListFiles(vcs);
19             case "exit" ->  new Exit(vcs);
20             default->{ System.out.println("unknown command " + cmdName);
21                 yield null;}
22         };
23     }
24 }
```

Listing 5: Exit.java

```
1  public final class Exit extends Command {
2
3      public Exit(VCS vcs) {
4          super(vcs);
5      }
6
7      @Override
8      public void execute() {
9          Util.exit();
10     }
11
12 }
```

Listing 6: ListFiles.java

```
1  public non-sealed class ListFiles extends Command {
2
3      public ListFiles(VCS vcs) {
4          super(vcs);
5      }
6
7      @Override
8      public void execute() {
9          for (String file: Util.listFiles(getVCS().getRootDir())) {
10             System.out.println(file);
11         }
12     }
13
14 }
```

Listing 7: Commit.java

```
1  public class Commit extends ListFiles implements Modifying {
2
3      public Commit(VCS vcs) {
4          super(vcs);
5      }
6
7      @Override
8      public void execute() {
9          String ts = Util.getTimestamp();
10         String backupDir = getVCS().getBackupDir();
11         String rootDir = getVCS().getRootDir();
12         String newDir = Util.appendFileOrDirname(backupDir, ts);
13         Util.mkdir(newDir);
14         for (String file: Util.listFiles(backupDir)) {
15             String src = Util.appendFileOrDirname(backupDir, file);
16             String dest = Util.appendFileOrDirname(newDir, file);
17             Util.moveFile(src, dest);
18         }
19         for (String file: Util.listFiles(rootDir)) {
20             String src = Util.appendFileOrDirname(rootDir, file);
21             String dest = Util.appendFileOrDirname(backupDir, file);
22             Util.copyFile(src, dest);
23         }
24         System.out.println("Committed the following files:");
25         super.execute();
26     }
27
28     @Override
29     public String getInformation() {
30         return "\nFiles: Copy and Move\nDirectory: create";
31     }
32 }
```

Listing 8: VCS.java

```
1  import java.io.*;
2  import java.util.*;
3
4  public class VCS {
```

```
5
6       private String rootDir;
7
8       public VCS(String dir) {
9           this.rootDir = dir;
10          if (!new File(getBackupDir()).exists()) {
11              Util.mkdir(getBackupDir());
12              System.out.println("initialized empty repository");
13          }
14      }
15
16      /**
17       * @return the backup directory of the version control system
18       */
19      public String getBackupDir() {
20          return Util.appendFileOrDirname(rootDir, "vcs");
21      }
22
23      /**
24       * @return the root directory of the version control system
25       */
26      public String getRootDir() {
27          return rootDir;
28      }
29
30  }
```

Listing 9: Util.java

```
1   import java.io.*;
2   import java.nio.file.*;
3   import java.util.*;
4
5   public class Util {
6
7       /**
8        * Creates a new file- or directory name by appending "fileOrDirname"
9        * to "dirname".Takes care of the operating-system specific
10       * file-separator which has to be in between
11       * (e.g., "/" on Linux, but "\" on Windows)
12       * @return a new file- or directory name, pointing to
13       * the "fileOrDirname" in the directory "dirname"
14       */
15      public static String appendFileOrDirname(String dirname, String fileOrDirname) {
16          return new File(dirname + File.separator + fileOrDirname).getAbsolutePath();
17      }
18
19      /**
20       * Creates the directory "dirname".
21       * Fails (and enforces termination) if creating "dirname" fails.
22       */
23      public static void mkdir(String dirname) {
24          if (!new File(dirname).mkdir()) {
25              System.err.println("error creating directory " + dirname);
26              System.exit(-1);
27          }
28      }
```

```
29
30      /**
31       * Moves the file "srcFilename" to "destFilename".
32       * Fails (and enforces termination) if moving fails
33       * (e.g., if "destFilename" already exists).
34       */
35      public static void moveFile(String srcFilename, String destFilename) {
36          try {
37              Files.move(new File(srcFilename).toPath(),
38                      new File(destFilename).toPath());
39          } catch (IOException e) {
40              System.err.println("error writing " + srcFilename
41                      + " to " + destFilename);
42              System.exit(-1);
43          }
44      }
45
46      /**
47       * Copies the file "srcFilename" to "destFilename".
48       * Fails (and enforces termination) if copying fails
49       * (e.g., if "destFilename" already exists).
50       */
51      public static void copyFile(String srcFilename, String destFilename) {
52          try {
53              Files.copy(new File(srcFilename).toPath(),
54                      new File(destFilename).toPath());
55          } catch (IOException e) {
56              System.err.println("error writing " + srcFilename
57                      + " to " + destFilename);
58              System.exit(-1);
59          }
60      }
61
62      /**
63       * @return a string which uniquely identifies a point in time
64       * (search for "Unix timestamp" for further information)
65       */
66      public static String getTimestamp() {
67          return Long.toString(System.currentTimeMillis());
68      }
69
70      /**
71       * @return the names of all files (excluding directories)
72       * contained in the directory "dirname"
73       */
74      public static String[] listFiles(String dirname) {
75          List<String> filenames = new ArrayList<>();
76          for (File f : new File(dirname).listFiles()) {
77              if (!f.isDirectory()) {
78                  filenames.add(f.getName());
79              }
80          }
81          return filenames.toArray(new String[filenames.size()]);
82      }
83
84      /**
```

```
85       * Terminates the program.
86       */
87      public static void exit() {
88          System.exit(0);
89      }
90
91  }
```

Listing 10: Main.java

```
1   import java.io.*;
2   import java.util.*;
3
4   public class Main {
5
6       /**
7        * @param args the only expected argument is the path to the root directory
8        */
9       public static void main(String[] args) {
10          if (args.length != 1) {
11              System.out.println("wrong number of arguments");
12          } else {
13              File f = new File(args[0]);
14              if (!f.exists()) {
15                  System.out.println(args[0] + " does not exist");
16              } else if (!f.isDirectory()) {
17                  System.out.println(args[0] + " is not a directory");
18              } else if (!f.canWrite()) {
19                  System.out.println(args[0] + " is read-only");
20              } else {
21                  VCS vcs = new VCS(args[0]);
22                  Scanner scanner = new Scanner(System.in);
23                  while (true) {
24                      System.out.print("> ");
25                      Command cmd = Command.parse(scanner.nextLine(), vcs);
26                      if (cmd != null) {
27                          if( cmd instanceof Modifying m){
28                             System.out.println("This command does the " +
29                                     "following modifiying operations: "
30                                     + m.getInformation());
31                          }
32                          cmd.execute();
33                      }
34                  }
35              }
36          }
37      }
38
39  }
```

Listing 11: Main.java

```
1   public interface Modifying {
2       public String getInformation();
3   }
```

Aufgabe 8 (Codescape): (Codescape)

Schließen Sie das Spiel Codescape ab, indem Sie die letzten Missionen von Deck 7 lösen. Genießen Sie anschließend das Outro. Dieses Deck enthält keine für die Zulassung relevanten Missionen.

Hinweise:

- Es gibt drei verschiedene Möglichkeiten wie die Story endet, abhängig von Ihrer Entscheidung im finalen Raum.
- Verraten Sie Ihren Kommilitonen nicht, welche Auswirkungen Ihre Entscheidung hatte, bevor diese selbst das Spiel abgeschlossen haben.

Lösung: