



HOCHSCHULE RUHR WEST  
UNIVERSITY OF APPLIED SCIENCES

# **Dateisysteme**

## **Dokumentation**

im Modul Betriebssysteme  
Studiengang Angewandte Informatik  
der Hochschule Ruhr West

## **Gruppe 4**

**Gian-Luca Afting; 10017447**

**Patrick Jansen; 10017514**

**Tim Preisler; 10017458**

Bottrop, September 2024

## Kurzfassung

Die Arbeit untersucht die Struktur der Datenhaltung auf Festplatten und stellt verschiedene Dateisysteme vor, die unterschiedliche Ansätze zur Verwaltung von Daten verfolgen. Im Rahmen des Projekts wurden diese Dateisysteme in einer Simulation implementiert und in einem benutzerfreundlichen Interface dargestellt. Der Schwerpunkt lag auf der verständlichen und analogen Umsetzung der Dateisysteme. Das Dokument beschreibt detailliert die Nutzung des User Interface sowie die Interaktion mit den Schnittstellen der implementierten Dateisysteme. Zwei spezifische Dateisysteme, FAT und INode, werden dabei näher erläutert und ihre Implementierung wird ausführlich dokumentiert.

# Inhaltsverzeichnis

<b>Kurzfassung .....</b>	<b>2</b>
<b>Inhaltsverzeichnis.....</b>	<b>3</b>
<b>Abbildungsverzeichnis.....</b>	<b>5</b>
<b>Tabellenverzeichnis.....</b>	<b>6</b>
<b>Abkürzungsverzeichnis.....</b>	<b>7</b>
<b>1     Einleitung .....</b>	<b>8</b>
<b>2     User Interface .....</b>	<b>9</b>
2.1    Speicher Wahl .....	9
2.2    Haupt Fenster .....	9
2.3    Fragmentierung .....	10
2.4    Kopieren .....	11
2.5    Löschen .....	12
2.6    Ordner Erstellen.....	12
2.7    Dateien Erstellen .....	13
2.8    Pfad eingaben.....	14
<b>3     Schnittstellen .....</b>	<b>15</b>
3.1    Schnittstelle: FileSystemInterface.....	15
3.1.1   Erstellen eines Dateisystems.....	15
3.1.2   Erstellen einer Datei für ein Dateisystem.....	16
3.1.3   Erstellen eines Ordners für ein Dateisystem .....	17
3.1.4   Löschen von Dateien und Ordner.....	17
3.1.5   Fragmentierung der Festplatte .....	18
3.1.6   Ausgabe von Metadaten.....	19
3.2    Schnittstelle: FolderInformationInterface .....	20
3.3    Schnittstelle: FileInformationInterface.....	21
<b>4     Implementierung des Dateisystems FAT .....</b>	<b>22</b>
4.1    FAT-System anlegen .....	22
4.2    FAT-Datei anlegen.....	23
4.3    FAT-Ordner anlegen.....	25
4.4    FAT-Dateien und Ordner löschen.....	26
4.4.1   FAT-Dateien löschen .....	27
4.4.2   FAT-Ordner löschen .....	28

<b>5</b>	<b>Implementierung des Dateisystems Inode .....</b>	<b>29</b>
5.1.1	Aufbau einer Inode.....	29
5.2	Inode-System anlegen .....	30
5.3	Inode Datei anlegen .....	31
5.3.1	Datenblockspeicherung .....	32
5.3.2	Zufällige Belegung .....	33
5.4	Inode-Ordner anlegen .....	34
5.5	Inode-Dateien und Ordner löschen .....	34
5.5.1	Inode-Dateien löschen .....	35
5.5.2	Inode-Ordner löschen .....	36

## Abbildungsverzeichnis

Abbildung 1: Auswahl der Speicherpartition .....	9
Abbildung 2: Dateisystem anlegen .....	15
Abbildung 3: Datei anlegen spezialisierte Methode .....	16
Abbildung 4: Datei anlegen ohne Zeiten .....	16
Abbildung 5: Datei kopieren .....	16
Abbildung 6: Ordner erstellen .....	17
Abbildung 7: Löschen einer Datei .....	17
Abbildung 8: Löschen eines Ordners .....	17
Abbildung 9: Fragmentierung einer Festplatte .....	18
Abbildung 10: Festplatte defragmentieren .....	18
Abbildung 11: Ausgabe der Festplattenblöcke .....	19
Abbildung 12: Blöcke setzen .....	19
Abbildung 13: Anzahl der genutzten Festplatte .....	19
Abbildung 14: Rückgabe der Ordner-Metadaten .....	20
Abbildung 15: Rückgabe der Datei-Metadaten .....	20
Abbildung 16: Ordnername ausgeben .....	20
Abbildung 17: Ordnergröße ausgeben .....	20
Abbildung 18: FAT Festplattengröße definieren .....	22
Abbildung 19: FAT initialisieren .....	23
Abbildung 20: FAT Dateiindexbestimmung .....	23
Abbildung 21: FAT Dateimetadaten pflegen .....	24
Abbildung 22: FAT-Blöcke reservieren .....	24
Abbildung 23: Eintrag ins DTF – Aufruf .....	24
Abbildung 24: "insertFile" des DTF .....	25
Abbildung 25: DTF im FAT-Dateisystem .....	25
Abbildung 26: formale Prüfung des Pfads in createFolder .....	26
Abbildung 27: FAT-Dateisystem Ordner anlegen .....	26
Abbildung 28: FAT - Dateilöschen .....	27
Abbildung 29: "remove_file" des DTF .....	27
Abbildung 30: FAT Blöcke freigeben .....	27
Abbildung 31: FAT-Ordner löschen .....	28
Abbildung 32: INode-System erstellen .....	30
Abbildung 33: INode-Datei anlegen .....	31
Abbildung 34: Datenblockspeicherung von INodes .....	32
Abbildung 35: Datenblockspeicherung der indirekten Blöcke .....	32
Abbildung 36: Bereitstellung eines freien Datenblocks .....	33
Abbildung 37: INode-Ordner anlegen .....	34
Abbildung 38: INode-Datei löschen .....	35
Abbildung 39: INode-Ordner löschen .....	36

## Tabellenverzeichnis

Tabelle 1: Methoden des FileInformationInterface .....	21
Tabelle 2: INode-Aufbau .....	29

## Abkürzungsverzeichnis

FAT	File Allocation Table
UI	User Interface
MBR	Master Boot Record
DTF	Directory Table Format
Inode	Index Node

# 1 Einleitung

Dieses Projekt beschäftigt sich mit der Implementierung und Simulation von zwei grundlegenden Dateisystemen: FAT und INode. Beide Dateisysteme wurden im Rahmen eines Projekts für das Modul „Betriebssysteme“ entwickelt, um die Funktionsweise und Struktur der Datenhaltung auf Festplatten praxisnah zu veranschaulichen. Neben der Implementierung dieser Dateisysteme wurde ein benutzerfreundliches User Interface entwickelt, das die Interaktion mit den Dateisystemen ermöglicht. Zusätzlich wurde eine CD-ROM simuliert, die nur einmal beschrieben werden kann, um die besonderen Herausforderungen dieses Speichermediums zu verdeutlichen.

Das zentrale Ziel dieser Dokumentation ist eine umfassende Erklärung des Programmcodes und ein Handbuch zur Verwendung von diesem zu bieten. Der besondere Fokus liegt auf der Nachvollziehbarkeit der Implementierung und der zugrunde liegenden Konzepte. Die Arbeit beschreibt detailliert die technische Umsetzung der Dateisysteme und das Design des User Interface.



## 2 User Interface

### 2.1 Speicher Wahl

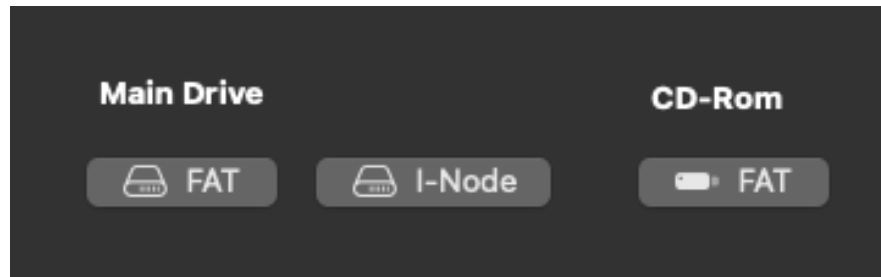


Abbildung 1: Auswahl der Speicherpartition

Mit der Wahl der Speicherpartition, wird sowohl die Partition gewählt und im nächsten Fenster angezeigt als auch das Dateisystem angelegt. Mehr dazu in 3.1.1

Simuliert, werden, auf der Hauptfestplatte zwei Partitionen, jeweils in Dateisystem FAT und INode formatiert und auf der CD-ROM eine FAT-Partition.

### 2.2 Haupt Fenster

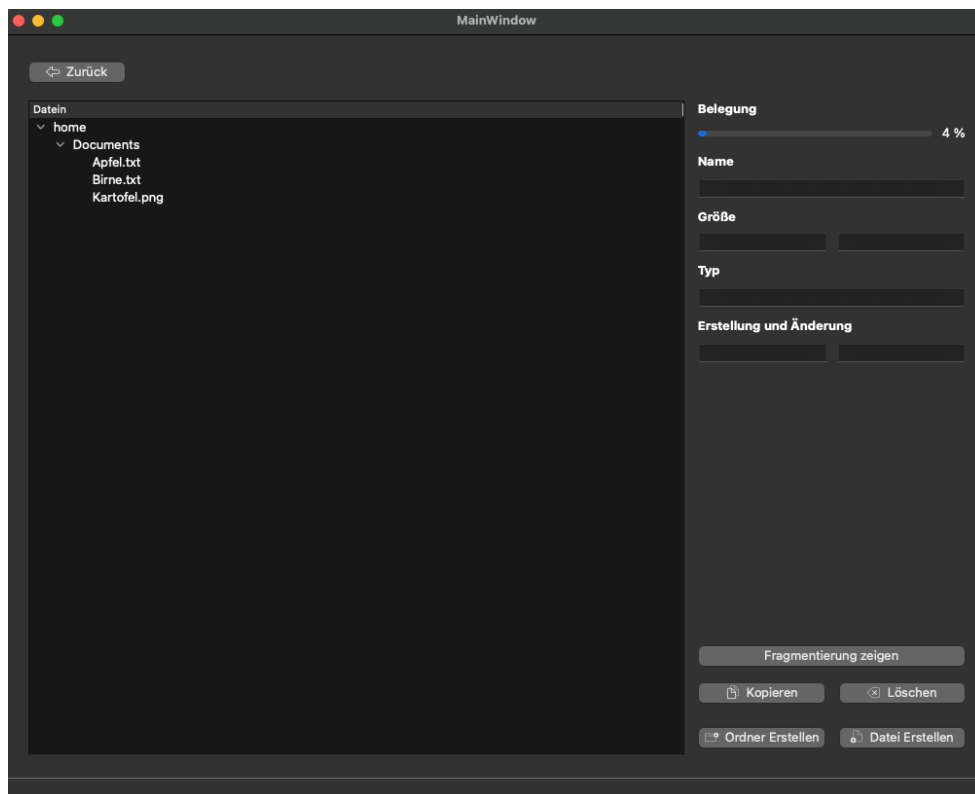


Abbildung 2: Haupt Fenster

Auf der linken Seite gibt es von oben an, zuerst einen Button, wodurch der User zurück auf die Auswahl der Partition gelangt. Gefolgt von einem Dateibaum, der die Dateien, auf der Partition mit samt ihrer Ordner Struktur anzeigt.

Der Baum ist intern mit dem QT eigenen „treeView“ Widget umgesetzt. Jedoch wurde das Treeltem und TreeModel, auf unsere Bedürfnisse angepasst. Die Items in der „tree-View“ sind in Form eines Baums gespeichert. So gibt es ein Wurzelement („root item“), dass in der UI nicht angezeigt wird, und die einzelnen Items haben alle einen Pointer auf ihr „parent“- und ihre „child“-Elemente.

Auf der rechten Seite gibt es von oben an:





1. Belegung: Hier wird der benutzte Speicher in Prozent und zusätzlich anhand eines Balkens angezeigt. In unserer Simulation gibt es 64 Blöcke mit jeweils 512 Byte, pro Partition. Dies wird im Code festgelegt.
2. Name: Zeigt den Namen der gewählten Datei.
3. Größe: Zeigt die Größe der gewählten Datei/Ordners in Bytes und Blöcken.
4. Typ: Zeigt den Dateitypen. In der Simulation gibt es die Dateitypen „png“ (Bilder), „txt“ (Textdateien) und „mp3“ (Tonträger).
5. Erstellung und Änderung: Zeigt Daten zur Erstellung und Änderung von Dateien.

## 2.3 Fragmentierung



Abbildung 3: Fragmentierungsansicht

In diesem Dialogfenster wird sowohl die aktuelle Fragmentierung in Prozent visualisiert als auch jeder einzelne Block, mit dem dazugehörigen Status:

- Ziffer   
Wird eine Ziffer angezeigt, so ist dieser Block belegt. Dabei repräsentiert die Ziffer die Indizes der Datei. Die Blöcke mit den gleichen Indizes gehören zusammen zu einer Datei.
- Frei   
Grüne Blöcke mit einem F sind als „Frei“ gekennzeichnet und können von einer neuen Datei belegt werden.
- Defekt   
Rote Blöcke mit einem X sind als „Defekt“ simuliert und werden vom System nicht benutzt.
- Reserviert   
Graue Blöcke mit einem R sind als „Reserviert“ simuliert und werden vom System nicht benutzt.

Blöcke, an denen nicht bereits eine Datei liegt, die also nicht blau mit einer Ziffer versehen sind, können angeklickt werden und somit kann durch die Zustände „Frei“, „Defekt“ und „Reserviert“ gewechselt werden.

Das Anklicken eines belegten Blocks gibt eine Fehlermeldung.

Wenn der Benutzer sich nicht auf der CD befindet, kann außerdem noch der Button Defragmentieren angewählt werden. Dieser ruft eine entsprechende Funktion zur Defragmentierung auf. Mehr dazu in 3.1.5.

## 2.4 Kopieren

Wenn eine Datei ausgewählt ist und der „Kopieren“-Knopf gewählt wird, öffnet sich der Kopierdialog.

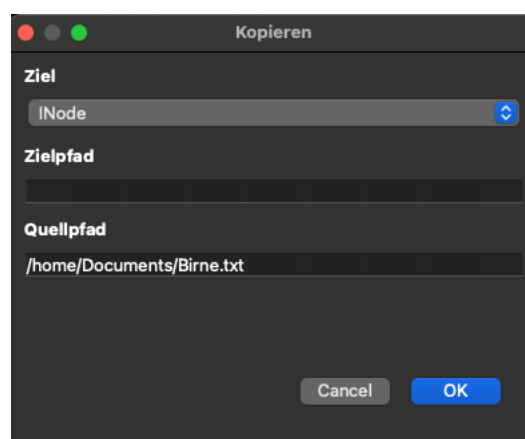


Abbildung 4: Kopierdialog

Beim Ziel können jeweils die anderen Partitionen als Ziel ausgewählt werden. Der Zielpfad ist der Pfad, auf der Zielpartition, zu der die Datei kopiert wird.

Der Quellpfad basiert auf der ausgewählten Datei.

## 2.5 Löschen

Ein Klick auf den Button „Löschen“, öffnet den Löschmodal. Dieser basiert auf der gewählten Datei und fragt ab, ob diese wirklich gelöscht werden soll. Dies geht ebenfalls mit Ordnern.

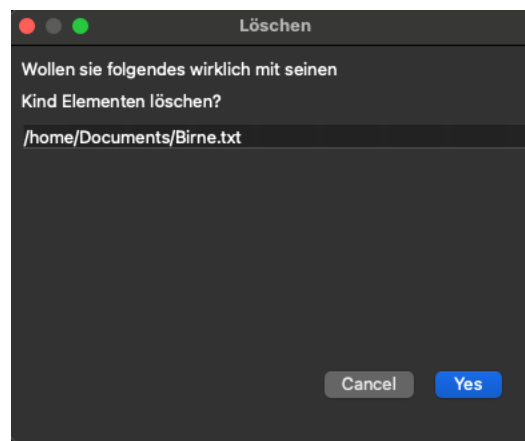


Abbildung 5: Löschmodal

Mehr zum Löschen unter 3.1.4.

## 2.6 Ordner Erstellen

Ein Klick auf den Button „Ordner Erstellen“ öffnet den Ordner Erstellungsdialog. Hier können neue Ordner erstellt werden. In dem Eingabefeld muss sowohl der Pfad als auch am Ende der Name des neuen Ordners eingefügt werden. Mehr zu Pfadfeldern im Abschnitt 2.8. Die Datei wird nur auf der aktuell gewählten Partition erstellt.

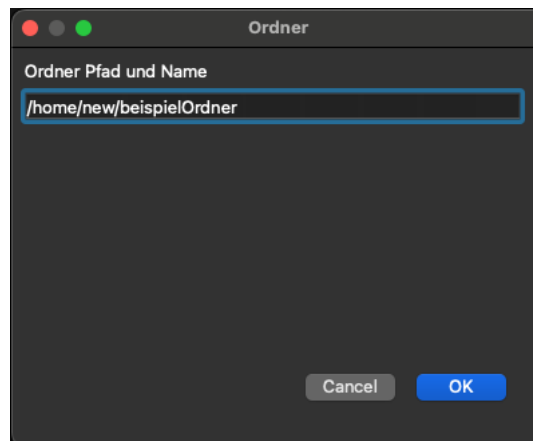


Abbildung 6: Ordner Dialog

Mehr zu der Erstellung von Ordnern unter: 3.1.3.

## 2.7 Dateien Erstellen

Ein Klick auf den Button „Datei erstellen“ öffnet den Dialog zum Erstellen von Dateien.

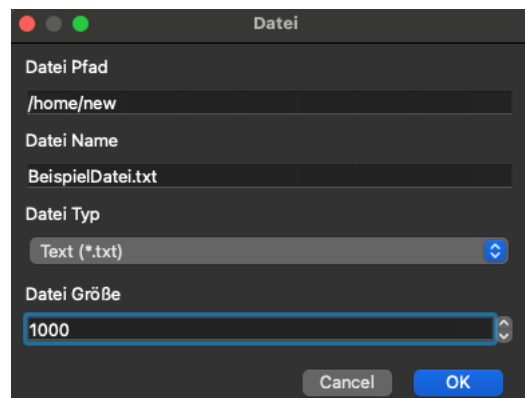


Abbildung 7: Ordner Dialog

Hier wird im obersten Feld der Pfad, an dem die Datei erstellt werden soll, gewählt. Mehr zu Pfadeingaben unter dem Abschnitt 2.8.

Darauffolgend kann der Dateiname eingegeben werden. Dies kann mit oder ohne Endung geschehen, da die Endung aus Sicherheitsgründen entfernt wird und die aus dem Select-Feld entnommen wird.

In dem Select-Feld lässt sich dann der Dateityp wählen. Hier kann zwischen txt (Textdatei), png (Bilddatei) und mp3 (Tonträger) gewählt werden.

Im letzten Feld lässt sich noch die Größe der simulierten Datei wählen.

Mehr zur Erstellung von Dateien in 3.1.2.

## 2.8 Pfad eingaben

Bei den Pfadeingabefeldern ist zu beachten, dass die Eingaben einigen Restriktionen unterstehen und anschließend die Eingabe ebenfalls überprüft wird. So sind nur die Zeichen A-Z, a-z, 0-9 und „/“ als Eingabe erlaubt. Außerdem wird vom System immer sichergestellt, dass ein „/“ am Anfang ist und keines am Ende, wenn dieses nicht benötigt wird. Es macht dementsprechend keinen Unterschied, ob ein Slash am Anfang und oder Ende steht. Es ist nur wichtig, dass die einzelnen Ordner mit „/“ getrennt sind.

Außerdem sollte beachtet werden, dass Ordner und Datei Namen Case-Sensitiv sind.

## 3 Schnittstellen

Eine Schnittstelle definiert die Kommunikationsstruktur zwischen zwei Systemkomponenten. Im Dateisystemprojekt wird diese Art der Schnittstelle verwendet, um die Kommunikation zwischen dem User Interface und der dahinterstehenden Logik zu ermöglichen. Die Kommunikation basiert dabei hauptsächlich auf drei abstrakten Klassen, die von den spezialisierten Klassen implementiert werden müssen. Die abstrakte Klasse „FileSystemInterface“ bildet dabei die Hauptschnittstelle, über die ein neues Dateisystem erstellt und verwendet werden kann. Die beiden weiteren abstrakten Klassen, „FileInformationInterface“ und „FolderInformationInterface“, definieren die Metadaten von Dateien und Ordnern, wodurch diese Informationen auch im User Interface angezeigt werden können. Im Folgenden wird die Implementierung der wichtigsten Methoden genauer erläutert sowie deren Notwendigkeit und Verwendung beschrieben.

### 3.1 Schnittstelle: FileSystemInterface

Die Zuständigkeit der Schnittstelle „FileSystemInterface“ liegt bei dem Kontrollieren eines Dateisystems. Dabei besteht die Möglichkeit ein neues Dateisystem zu erstellen, Dateien anzulegen oder zu löschen.

Bei der Anwendung der Methoden ist zu beachten, dass ausgewählte Methoden einen Rückgabewert des Datentyps Boolean aufweisen. Durch diese Rückgabe wird realisiert auszugeben, ob ein Vorgang erfolgreich war. Wenn der Rückgabewert auf „true“ gesetzt wird, war der Vorgang erfolgreich, wenn dem nicht so ist, wird der Rückgabewert „false“ geschrieben.

#### 3.1.1 Erstellen eines Dateisystems

Um ein Dateisystem anzulegen, wird die Methode „createSystem“ genutzt. Dabei müssen die Anzahl der Blöcke und die Blockgröße auf der Festplatte mit angegeben werden.

```
virtual void createSystem(size_t diskSize, size_t sizeBlock) = 0;
```

Abbildung 2: Dateisystem anlegen

Diese Methode bewirkt im Hintergrund, dass der entsprechende Speicher für gewisse Komponenten reserviert wird und die grundlegende Struktur des Dateisystems angelegt wird.

### 3.1.2 Erstellen einer Datei für ein Dateisystem

Grundlegend wird eine Datei auf einem Weg angelegt werden. Dafür ist die Methode „createFile“ angelegt werden.

```
virtual bool createFile(size_t szFile,
                        std::string path,
                        std::string fileend,
                        std::string filekind,
                        std::string permissions,
                        std::string owner,
                        long long modification_time,
                        long long creation_time) = 0;
```

Abbildung 3: Datei anlegen spezialisierte Methode

Als Übergabeparameter werden insbesondere einige Metadaten übergeben. Dabei werden die Dateigröße, die Dateiendung, die Dateiart, Berechtigungen, Benutzer und die zwei Zeiten übergeben. Des Weiteren beschreibt der Dateipfad den gesamten Pfad der Datei exklusive des Dateinamens. Diese Methode sollte nicht vom Nutzer direkt aufgerufen werden, da bei dieser Methode die „modification\_time“ und die „creation\_time“ manuell festgelegt werden muss. Aus diesem Grund wird die Methode intern von verschiedenen Methoden aufgerufen, welche für den Nutzer zur Verfügung stehen.

Um eine Datei neu anzulegen wird dafür die überladene Methode „createFile“ genutzt. Dabei wird die „modification\_time“ und die „creation\_time“ automatisch gepflegt, sodass der Nutzer diese nicht manipulieren kann. Die Schnittstelle sieht wie folgt aus:

```
virtual bool createFile(size_t szFile,
                        std::string path,
                        std::string fileend,
                        std::string filekind,
                        std::string permissions,
                        std::string owner) = 0;
```

Abbildung 4: Datei anlegen ohne Zeiten

Bei der Implementierung dieser Datei wird die umfangreichere spezifizierte Methode (siehe Abbildung 2) aufgerufen und die Zeiten automatisch gepflegt.

Zuletzt kann eine Datei auf einem Dateisystem ebenfalls angelegt werden, indem diese von einem anderen Dateisystem hineinkopiert wird. Um dies zu realisieren wird die Methode „copyFileToFileSystem“ aufgerufen. Dafür müssen drei Übergabeparameter gepflegt werden:

```
virtual bool copyFileToFileSystem(std::string pathToCopy,
                                  FileInformationInterface* fileInformations,
                                  FileSystemInterface* fileSystem) = 0;
```

Abbildung 5: Datei kopieren



Der Übergabeparameter „pathToCopy“ ist der Zielpfad, bei dem die Datei abgelegt werden soll. Des Weiteren beschreibt das „FileInformationInterface“ die Datei, welche kopiert werden soll. Zuletzt muss noch das Dateisystem angegeben werden, zu welchem kopiert werden soll. Da eine Datei intern im Wesentlichen durch das „FileInformationInterface“ beschrieben werden kann, wird bei der Kopierfunktion die umfangreiche spezifizierte Methode des Zieldateisystems aufgerufen.

### 3.1.3 Erstellen eines Ordners für ein Dateisystem

Um einen Ordner zu erstellen, gibt es die Möglichkeit die Methode „create\_folder“ aufzurufen. Diese Methode bekommt einen Übergabeparameter:

```
virtual bool createFolder(std::string path) = 0;
```

Abbildung 6: Ordner erstellen

Der Übergabeparameter beschreibt dabei den gesamten Pfad inklusive des neuen Ordernamens (Beispiel: „/home/user/documents/“).

Eine weitere Möglichkeit einen Ordner zu erstellen ist über die createFile-Methode. Dabei muss ebenfalls ein Pfad zu der Datei angegeben werden. Wenn in diesem Pfad Ordner angegeben werden, welche noch nicht vorhanden sind, werden diese implizit mit angelegt.

### 3.1.4 Löschen von Dateien und Ordner

Die Löschung einer Datei kann durch die Methode „removeFile“ realisiert werden:

```
virtual bool removeFile(std::string path) = 0;
```

Abbildung 7: Löschen einer Datei

Dabei muss der Pfad der Datei, inklusive des Dateinamens und Endung, angegeben werden. Dadurch wird die Datei vollständig und endgültig aus dem System gelöscht.

Analog zur Löschung einer Datei gibt es eine Methode zum Löschen eines Ordners.

```
virtual bool removeFolder(std::string path) = 0;
```

Abbildung 8: Löschen eines Ordners

Der Übergabeparameter „path“ bekommt dabei den Pfad zum zu löschenden Ordner übergeben (Beispiel: „/home/user/documents/“). Dabei wird immer der letzte erwähnte Ordner im Pfad gelöscht. Ebenfalls gelöscht, wird der gesamte Inhalt des Ordners.

### 3.1.5 Fragmentierung der Festplatte

Fragmentierung einer Festplatte beschreibt den Zustand, in dem die Daten auf der Festplatte nicht zusammenhängend, sondern in einzelnen, verstreuten Blöcken (Fragmenten) gespeichert werden. Dies passiert, wenn Dateien gelöscht, geändert oder neu gespeichert werden, da der freigewordene Speicherplatz oft nicht zusammenhängend ist. Neue Dateien werden daraufhin in diesen verstreuten Bereichen gespeichert, was dazu führt, dass eine Datei in mehrere Teile (Fragmente) aufgeteilt wird. Bei einer Hard Drive Disk sorgt diese Fragmentierung für langsamere Lese- und Schreibvorgänge, da das Speichermedium eine größere Strecke zwischen einzelnen Datenblöcken zurücklegen muss. Um sich den Wert der Fragmentierung ausgeben zu lassen gibt es die Methode „getFragmentation“.

```
virtual float getFragmentation() = 0;
```

Abbildung 9: Fragmentierung einer Festplatte

Dabei wird der Wert der Fragmentierung als float zurückgegeben, wobei „0%“-Fragmentierung bedeuten würde, dass keine Fragmente vorhanden sind und alle Dateien zusammenhängen und „100%“-Fragmentierung würde bedeuten, dass alle Blöcke aller Dateien nur als Fragmente vorhanden sind.

Um eine mögliche hohe Fragmentierung zu optimieren, gibt es die Methode „defragDisk“. Dabei wird versucht die Blöcke einer Datei möglichst zusammenhängend zu gestalten, sodass die Lese- und Schreibzeiten schneller wären.

```
virtual void defragDisk() = 0;
```

Abbildung 10: Festplatte defragmentieren

Durch diese Methode kann nicht gewährleistet werden, dass die Festplatte eine „0%“-Fragmentierung aufweist, aber es wird algorithmisch versucht diesem nahe zu kommen. Den Status der einzelnen Blöcke kann durch die Methode „getBlockStates“ ausgegeben werden.

```
virtual std::map<int, std::pair<BlockState, int>> getBlockStates() = 0;
```

Abbildung 11: Ausgabe der Festplattenblöcke

Die Rückgabe der Methode wird als Map realisiert. Dabei wird ein Index, dargestellt durch den Datentyp Integer, auf ein Blockstate und eine mögliche Dateiidentifikationsnummer aufgezeigt. Die Dateiidentifikationsnummer hat für die Blockstatus „FREE“, „RESERVED“ und „DEFECT“ den Wert „-1“ zugewiesen, da diese keine Dateizugehörigkeit aufweisen können. Der Blockstatus „OCCUPIED“ hat hingegen eine Dateiidentifikationsnummer zugewiesen.

Zuletzt kann der Status eines Blocks manuell auf „FREE“, „RESERVED“ oder „DEFECT“ gesetzt werden, um die Fragmentierung besser simulieren zu können. Dabei kann ein freier Block auf „RESERVED“ oder „DEFECT“ gesetzt werden und ebenfalls können Blöcke, welche den Status „RESERVED“ oder „DEFECT“ aufweisen auf den Status „FREE“ gesetzt werden. Diese Funktion kann durch die Methode „setBlock“ ausgeführt werden.

```
virtual bool setBlock(int blockIndex, BlockState blockstate) = 0;
```

Abbildung 12: Blöcke setzen

Diese Methode bekommt zum einen den Blockindex übergeben, welcher angepasst werden soll. Zum anderen wird ein Blockstatus übergeben, welcher der ausgewählte Block erhalten soll. Dabei ist zu beachten, dass nur die obigen Richtungen erlaubt sind.

### 3.1.6 Ausgabe von Metadaten

Um einige Metadaten zu der Festplatte, einem Ordner oder einer Datei zu lesen gibt es dafür ausgewählte Methoden.

Für die Festplatte ist es möglich den genutzten Speicher ausgeben zu lassen. Dafür wird die Methode „getPartitionUsage“ aufgerufen.

```
virtual size_t getPartitionUsage() = 0;
```

Abbildung 13: Anzahl der genutzten Festplatte

Zurückgeben wird die Anzahl der Bytes, welche auf der Festplatte bereits genutzt werden.

Des Weiteren stehen einige Metadaten zu einem Ordner zur Verfügung. Diese stehen gebündelt unter „getFolder“ zur Verfügung.

```
virtual FolderInformationInterface *getFolder(std::string path) = 0;
```

Abbildung 14: Rückgabe der Ordner-Metadaten

Um einen Rückgabewert zu erhalten, muss der Pfad zu dem Ordner analog zur Löschung oder Erstellung übergeben werden. Als Rückgabewert wird die Schnittstelle „FolderInformationInterface“ zurückgegeben, welche jegliche Metadaten eines Ordners enthält.

Analog zum Ordner können ebenfalls die Metadaten einer Datei zurückgegeben werden. Dafür steht die Methode „getFile“ zur Verfügung.

```
virtual bool removeFile(std::string path) = 0;
```

Abbildung 15: Rückgabe der Datei-Metadaten

Um einen Rückgabewert zu erhalten, muss der Pfad zu der Datei analog zur Löschung übergeben werden. Als Rückgabewert wird die Schnittstelle „FileInformationInterface“ zurückgegeben, welche jegliche Metadaten einer Datei enthält.

## 3.2 Schnittstelle: FolderInformationInterface

Um die Metadaten zu einem Folder auslesen zu können, gibt es die Schnittstelle FolderInformationInterface. Darin gibt es zwei Methoden um den Ordnername oder die Größe des Ordner abzufragen.

Für den Name des Ordners gibt es die Methode „get\_folder\_name“. Diese gibt einen String zurück, welcher den Ordnername repräsentiert.

```
virtual std::string get_folder_name() = 0;
```

Abbildung 16: Ordnername ausgeben

Des Weiteren ist es möglich die Größe eines Ordners in Byte ausgegeben zu bekommen. Dabei werden alle enthaltenen Ordner und Dateien zusammengezählt. Dies geschieht mit der Methode „get\_folder\_size“.

```
virtual size_t get_folder_size() = 0;
```

Abbildung 17: Ordnergröße ausgeben

Als Rückgabewert gibt es einen Wert im Datentyp "size\_t". Dieser repräsentiert die Byte die der Ordner groß ist.

### 3.3 Schnittstelle: FileInformationInterface

Analog zu dem FolderInformationInterface weist das FileInformationInterface einige Methoden zum Auslesen der Metadaten einer Datei auf. Durch diese Methoden können folgende Metadaten ausgelesen werden:

Metadaten	Methode	Rückgabewert
Dateiname	<code>virtual std::string get_file_name() = 0;</code>	String mit Dateina-men
Dateiart	<code>virtual std::string get_file_kind() = 0;</code>	String mit Dateiart
Dateiendung	<code>virtual std::string get_file_prefix() = 0;</code>	String mit Dateien-dung
Berechtigungen	<code>virtual std::string get_file_permissions() = 0;</code>	String mit Berechti-gungen
Inhaber	<code>virtual std::string get_file_owner() = 0;</code>	String mit Dateiin-haber
Änderungsdatum	<code>virtual long long get_file_modification_time() = 0;</code>	Long long mit Än-derungsdatum
Erstelldatum	<code>virtual long long get_file_creation_time() = 0;</code>	Long long mit Er-stelldatum
Dateigröße	<code>virtual size_t get_file_size() = 0;</code>	Size_t mit Dateig-röße

Tabelle 1: Methoden des FileInformationInterface

## 4 Implementierung des Dateisystems FAT

Ein mögliches gängiges Dateisystem ist die File Allocation Table (FAT). Es verwaltet die Speicherung von Daten auf einem Datenträger, indem es eine Tabelle verwendet, die Informationen darüber speichert, wo Dateien auf dem Medium abgelegt sind. Analog zu diesem Vorgehen ist die Simulation ebenfalls aufgebaut. Im folgenden Abschnitt werden die wichtigsten Methoden und deren Implementierung erläutert.

### 4.1 FAT-System anlegen

Um ein Dateisystem anlegen zu können, werden einige Informationen benötigt. Zum einen wird benötigt, wie groß das Dateisystem werden soll. Zum anderen wird benötigt, wie groß ein Block sein soll. Zuletzt wird auch benötigt, wie viele Blöcke es im Allgemeinen gibt. Als Übergabeparameter reichen dabei zwei der Werte, da der dritte Berechnet werden kann.

```

BsFat *BsFat::createBsFat(size_t plattengroesse, size_t blockgroesse) {
    // Überprüfen, ob die Blockgröße nicht null ist, um eine Division durch null zu vermeiden
    if (blockgroesse == 0) {
        qCritical() << "Fehler: Die Blockgröße darf nicht null sein.";
    } else {

        // Berechnen der Anzahl der Blöcke
        size_t anzahlBlöcke = (plattengroesse + blockgroesse - 1) / blockgroesse;

        // Speicher für die BsFat-Struktur allokalieren
        auto *bsFat = new BsFat();
        if (bsFat == nullptr) {
            qCritical() << "Fehler: Speicher konnte nicht allokiert werden.";
        } else {

            // Speicher für das Blockstatus-Array allokalieren
            bsFat->fileAllocationTable = new FileAllocationTableRow[anzahlBlöcke];
            if (bsFat->fileAllocationTable == nullptr) {
                free(bsFat); // BsFat-Struktur freigeben, wenn die Allokation fehlschlägt
                qCritical() << "Fehler: Speicher für Blockstatus-Array konnte nicht allokiert werden.";
            } else {

```

Abbildung 18: FAT Festplattengröße definieren

Auf der obigen Abbildung ist zu erkennen, dass die „plattengroesse“ und „blockgroesse“ übergeben werden. Außerdem wird das entsprechende Array instanziiert und die Größe entsprechend auf die Anzahl der Blöcke gesetzt.

Zuletzt muss der Inhalt des Dateisystems initialisiert werden. Dafür muss das Directory Table Format (DTF) aufgesetzt werden und die Dateien vorbereitet werden. Das DTF ist zuständig für die Verzeichnisstruktur innerhalb eines FAT-Dateisystems.

```

bsFat->directoryTableFormat = new BsDtf("root");

// Initialisieren der BsFat-Struktur
bsFat->diskSize = plattengroesse;
bsFat->blockSize = blockgroesse;
bsFat->blockCount = anzahlBloেকে;

// Initialisieren aller Blöcke als frei
for (size_t i = 0; i < anzahlBloেকে; i++) {
    bsFat->fileAllocationTable[i].cleanRow();
}
bsFat->files = new BsFile[maximum_file_count];
for (int i = 0; i < maximum_file_count; i++) {
    bsFat->files[i].set_filename((char *) "\0");
}

return bsFat;

```

Abbildung 19: FAT initialisieren

Es ist zu erkennen, dass die File Allocation Table initialisiert wird und die Blöcke als frei eingetragen werden und das DTF mit dem Namen „root“ initialisiert wird.

## 4.2 FAT-Datei anlegen

Die Erstellung einer Datei im FAT-Dateisystem besteht im Wesentlichen aus 3 Schritten. Zum einen müssen die Metadaten abgespeichert werden. Des Weiteren ist es wichtig die entsprechenden Blöcke zu reservieren und die Datei im DTF einzutragen.

```

if (pFat == nullptr) {
    qCritical() << "Fehler: Ungültige Eingabe.";
    return false;
} else {
    size_t freeSpace = pFat->getFreeDiskSpace();
    if (freeSpace <= (size_t) szFile) {
        qWarning() << "Fehler: Nicht genügend freier Speicherplatz verfügbar.";
        return false;
    } else {
        int fileIndex = -1;
        for (int i = 0; i < maximum_file_count; i++) {
            if (pFat->get_files()[i].filename[0] == '\0') {
                fileIndex = i;
                break;
            }
        }

        if (fileIndex == -1) {
            qCritical() << "Fehler: Kein freier Platz für neue Datei verfügbar.";
            return false;
        } else {

```

Abbildung 20: FAT Dateindexbestimmung

Hier ist zu erkennen, dass zum eine formale Prüfungen durchgeführt werden, zum anderen wird der Index bestimmt, an welchem die neue Datei im Dateiarrray eingefügt werden soll.

```

std::vector<std::string> filePath = BsDtf::split(filename, '/');
pFat->get_files()[fileIndex].path = filename;
pFat->get_files()[fileIndex].filename = filePath[filePath.size() - 1];
pFat->get_files()[fileIndex].filePrefix = fileend;
pFat->get_files()[fileIndex].filekind = filekind;
pFat->get_files()[fileIndex].file_permissions = permissions;
pFat->get_files()[fileIndex].file_owner = owner;
pFat->get_files()[fileIndex].szFile = szFile;
pFat->get_files()[fileIndex].file_creation_time = modification_time;
pFat->get_files()[fileIndex].file_modification_time = creation_time;

pFat->get_files()[fileIndex].bsCluster = nullptr;

```

Abbildung 21: FAT Dateimetadaten pflegen

Im darauffolgenden ELSE-Zweig werden die Metadaten gepflegt. Dabei werden die übergebenen Metadaten gepflegt. Besonders zu betrachten ist der Pfad und Dateiname, da diese aus dem gesamten Pfad herausgelesen werden.

```

int blocksNeeded = (int) ((szFile + pFat->get_block_size() - 1) / pFat->get_block_size());
int blocksReserved = 0;
srand((unsigned int) time(nullptr));
for (int i = 0; blocksReserved < blocksNeeded; i < pFat->get_block_count() ? i++ : 0) {
    if (pFat->get_block_states()[i].getBlockState() == FREE) {
        if (pFat->get_files()[fileIndex].bsCluster == nullptr) {
            pFat->get_files()[fileIndex].bsCluster = new BsCluster();
            if (pFat->get_files()[fileIndex].bsCluster == nullptr) {
                qCritical() << "Fehler: Speicher für bsCluster konnte nicht allokiert werden.";
                return false;
            }
            pFat->get_files()[fileIndex].bsCluster->set_index(i);
            pFat->get_block_states()[i].setBsCluster(*pFat->get_files()[fileIndex].bsCluster);
        } else {
            pFat->get_block_states()[i].setBsCluster(
                BsCluster::appendBsCluster(pFat->get_files()[fileIndex].bsCluster, i));
        }
        blocksReserved++;
    }
}

```

Abbildung 22: FAT-Blöcke reservieren

Im nächsten Schritt werden die entsprechenden Blöcke in der FileAllocationTable reserviert und die Blöcke in der Datei eingefügt. Die Blöcke werden dabei als „bsCluster“ präsentiert. Dies ist eine doppelt verkettete Liste, in welcher die Daten gespeichert werden würden.

Der letzte Schritt besteht aus eintragen in die DTF. Dies ist wichtig, damit die Datei im richtigen Verzeichnis angezeigt wird.

```

pFat->get_dtf()->insertFile(filename, &pFat->get_files()[fileIndex]);

```

Abbildung 23: Eintrag ins DTF – Aufruf



Die Methode „insertFile“ des DTF ist dabei rekursiv aufgebaut:

```
void BsDtf::insertFile(std::string path, BsFile *entry) {
    std::vector<std::string> pathVector = split(path, '/');
    if (pathVector.size() == 1) {
        files.push_back(entry);
    } else {
        std::string dtf_name = pathVector[0];
        for (int i = 0; i < folder.size(); i++) {
            if (!folder[i]->folder_name.compare(dtf_name)) {
                pathVector.erase(pathVector.begin());
                folder[i]->insertFile(join(pathVector), entry);
                return;
            }
        }
        pathVector.erase(pathVector.begin());
        BsDtf *insertBsDtf = new BsDtf(dtf_name);
        folder.push_back(insertBsDtf);
        insertBsDtf->insertFile(join(pathVector), entry);
    }
}
```

Abbildung 24: "insertFile" des DTF

Als Übergabeparameter bekommt diese Methode den gesamten Pfad der Datei und die Datei selbst. Zu Beginn wird formal geprüft, ob diese im „root“-Verzeichnis abgelegt werden soll. Wenn dem so ist, dann wird diese direkt dort abgelegt, wenn dem nicht so ist, wird geprüft, ob der Ordner (beziehungsweise der erste Ordner im Pfad) vorhanden ist in welcher die Datei abgelegt werden soll. Sollte dies der Fall sein, wird dieser aufgerufen und rekursiv „insertFile“ aufgerufen mit dem Pfad der Datei ohne diesen Ordner. Damit wird rekursiv der Pfad durchgegangen. Sollte der Ordner nicht vorhanden sein, wird dieser angelegt, woraufhin ebenfalls wieder rekursiv „insertFile“ aufgerufen wird.

### 4.3 FAT-Ordner anlegen

Ein Ordner wird im Dateisystem „FAT“ durch die Methode „createFolder“ angelegt. Diese befindet sich im DTF, da dieses die Verzeichnisstruktur widerspiegelt. Es enthält alle Dateien und ist als Ordnerstruktur angelegt. Dies lässt sich anhand der privaten Attribute der DTF-Klasse ablesen:

```
private:
    std::vector<BsFile *> files;
    std::vector<BsDtf *> folder;

    std::string folder_name;
```

Abbildung 25: DTF im FAT-Dateisystem

Es ist zu erkennen, dass zum einen die Dateien in einem DTF-Objekt abgespeichert werden, aber auch weitere „BsDtf“-Objekte, wodurch die Verzeichnisstruktur aufgebaut wird.

Um nun einen neuen Ordner anzulegen, muss auf der gewünschten Ebene ein neues DTF-Objekt in eben diese Liste eingetragen werden. Dies wird realisiert, indem an die „createFolder“-Methode innerhalb des DTF der Pfad übergeben wird. Zu Beginn wird dieser formal geprüft, sodass dieser gültig ist.

```
bool BsDtf::createFolder(const std::string &path) {
    std::vector<std::string> pathVector = split(path, '/');

    if (pathVector.size() == 1) {
        if (hasFolder(pathVector[0])) {
            qCritical() << "Folder already created!";
            return false;
        }
    }
}
```

Abbildung 26: formale Prüfung des Pfads in createFolder

Wenn dies sichergestellt ist, wird die richtige Verzeichnisebene gesucht. Dies geschieht indem der Pfad anhand der „/“-Zeichen gesplittet wird und jeweils der erste Teil betrachtet wird. Es wird jeweils der erste Teil des Pfades betrachtet. Ist im aktuellen Verzeichnis ein Ordner mit diesem Namen vorhanden, wird dieser gewählt und darauf rekursiv erneut „createFolder“ aufgerufen. Sollte dies nicht der Fall sein, wird ein neuer Ordner angelegt. Daraufhin wird dieser ebenfalls gewählt und rekursiv „createFolder“ aufgerufen. Dieser Prozess wird bis zum Ende des Pfades fortgeführt.

```
} else {
    if (hasFolder(pathVector[0])) {
        BsDtf *insertIntoFolder = getFolder(pathVector[0]);
        if(insertIntoFolder == nullptr){
            return false;
        }
        pathVector.erase(pathVector.begin());
        if(!insertIntoFolder->createFolder(join(pathVector))){
            return false;
        }
    } else {
        folder.push_back(new BsDtf(pathVector[0]));
        BsDtf *insertIntoFolder = getFolder(pathVector[0]);
        if(insertIntoFolder == nullptr){
            return false;
        }
        pathVector.erase(pathVector.begin());
        if(!insertIntoFolder->createFolder(join(pathVector))){
            return false;
        }
    }
}
return true;
```

Abbildung 27: FAT-Dateisystem Ordner anlegen

## 4.4 FAT-Dateien und Ordner löschen

Um eine Datei oder ein Ordner aus dem FAT Dateisystem und zu löschen, müssen die oben genannten Schritte (Abschnitt 4.2) rückgängig gemacht werden. Dies besteht bei Dateien aus 2 Schritten und bei Ordnern wieder rekursiv.

#### 4.4.1 FAT-Dateien löschen

```
bool BsFatAdapter::removeFile(std::string path) {
    if(!bs_fat->get_dtf()->remove_file(path)){
        return false;
    }
    std::vector<std::string> pathVector = BsDtf::split(path, '/');
    return BsFile::deleteFile(bs_fat, pathVector[pathVector.size() - 1]);
}
```

Abbildung 28: FAT - Dateilöschen

Als erstes muss die Datei aus dem DTF gelöscht werden. Dafür wird erneut die Verzeichnisstruktur rekursiv durchiteriert, woraufhin der Eintrag mit der Methode „removeFileByName“ gelöscht wird.

```
bool BsDtf::remove_file(std::string filePath) {
    std::vector<std::string> pathVector = BsDtf::split(filePath, '/');
    if (pathVector.size() == 1) {
        this->removeFileByName(pathVector[0]);
    } else {
        std::string folder_name = pathVector[0];
        pathVector.erase(pathVector.begin());
        this->getFolderByName(folder_name)->remove_file(join(pathVector));
    }
}
```

Abbildung 29: "remove\_file" des DTF

Nachdem die Datei aus dem DTF gelöscht wurde, können die entsprechenden Blöcke freigegeben werden.

```
bool BsFile::deleteFile(const BsFat *pFat, std::string fileName) {
    int fileIndex = -1;
    for (int i = 0; i < maximum_file_count; i++) {
        if (pFat->get_files()[i].filename == fileName) {
            fileIndex = i;
            break;
        }
    }

    if (fileIndex == -1) {
        qCritical() << "Fehler: Datei konnte nicht gefunden werden.";
        return false;
    } else {
        while (pFat->get_files()[fileIndex].bsCluster->get_prev() != nullptr) {
            pFat->get_files()[fileIndex].bsCluster = pFat->get_files()[fileIndex].bsCluster->get_prev();
        }
        while (pFat->get_files()[fileIndex].bsCluster != nullptr) {
            pFat->get_block_states()[pFat->get_files()[fileIndex].bsCluster->get_index()].cleanRow();
            BsCluster *next = pFat->get_files()[fileIndex].bsCluster->get_next();
            free(pFat->get_files()[fileIndex].bsCluster);
            pFat->get_files()[fileIndex].bsCluster = next;
        }

        pFat->get_files()[fileIndex].filename = "\0";
        pFat->get_files()[fileIndex].filePrefix = "\0";
        pFat->get_files()[fileIndex].filekind = "\0";
        pFat->get_files()[fileIndex].file_permissions = "\0";
        pFat->get_files()[fileIndex].file_owner = "\0";
        pFat->get_files()[fileIndex].file_modification_time = 0;
        pFat->get_files()[fileIndex].file_creation_time = 0;
    }

    return true;
}
```

Abbildung 30: FAT Blöcke freigeben

In dieser Methode wird zu Beginn der Dateiindex der zu löschenden Datei gesucht. Aufgrund dieses Indizes wird zu Beginn durch die Blöcke der Datei iteriert und diese werden freigegeben. Daraufhin werden die Metadaten der Datei initialisiert.

#### 4.4.2 FAT-Ordner löschen

```
bool BsFatAdapter::removeFolder(std::string path)
{
    std::vector<std::string> pathVector = BsDtf::split(path, '/');
    if(pathVector.size() == 0 ){
        qCritical() << "Der Root-Folder kann nicht gelöscht werden!";
        return false;
    }

    std::string removeFolderName = pathVector[pathVector.size() - 1];
    pathVector.pop_back();

    BsDtf* folderOver;
    if(pathVector.size() == 0){
        folderOver = bs_fat->get_dtf();
    } else {
        folderOver = bs_fat->get_dtf()->getFolderByPath(BsDtf::join(pathVector));
    }

    if(folderOver == nullptr){
        return false;
    }

    if(!folderOver->getFolderByName(removeFolderName)->remove_dtf(bs_fat)){
        return false;
    }
    return folderOver->removeFolderByName(removeFolderName);
}
```

Abbildung 31: FAT-Ordner löschen

Um einen Ordner zu löschen muss dieser und all seine Inhalte aus dem DTF gelöscht werden.

Dafür wird zu Beginn der Pfad in seine Einzelteile zerlegt und es wird geprüft, ob der User versucht den root-Ordner zu löschen. Sollte dies der Fall sein, wird eine Fehlermeldung ausgelöst.

Daraufhin wird der Ordner gesucht, in welchem sich der zu löschende Ordner befindet, da von diesem aus das Löschen initiiert werden muss. Wenn dieser gefunden wurde, werden die Inhalte des zu löschenden Ordners gelöscht in dem die Methode „remove\_dtf“ auf dem Ordner aufgerufen wird. Diese löscht erneut rekursiv alle Inhalte des Ordners. Wenn dies erfolgreich war, wird der Ordner selbst gelöscht, in dem „removeFolderbyName“ durch den Ordner über dem zu löschenden Ordner aufgerufen wird.

## 5 Implementierung des Dateisystems INode

### 5.1.1 Aufbau einer INode

Ein weiteres mögliches gängiges Dateisystem ist die Index Node (INode).

Diese ist wie folgt aufgebaut:

Index	Die eindeutige Identifizierung einer INode	
Metadaten	INodetyp	Datei oder Ordner
	Dateiendung	Bspw.: .txt, .png
	Dateiart	Bspw. Text, Bild
	Besitzer	Der Besitzer
	Rechte	Berechtigung für Datei/Ordner
	Größe	Die Datei/Ordner-Größe
	Bearbeitungsdatum	Das Datum einer Bearbeitung
	Erstellungsdatum	Das Datum einer Bearbeitung
Direkt	12 Direkte Pointer	
Einfach-Indirekt	Einfacher Indirekter Pointer	
Zweifach- Indirekt	Zweifacher Indirekter Pointer	
Dreifach-Indirekt	Dreifacher Indirekter Pointer	

Tabelle 2: INode-Aufbau

Es wird bei INodes kein direkter Unterschied zwischen einer Datei und einem Ordner gemacht. Beide folgen demselben schematischen Aufbau. Bis auf den Unterschied, dass ein Ordner zusätzlich Kindelemente speichert.

## 5.2 INode-System anlegen

Um ein Dateisystem anlegen zu können, werden einige Informationen benötigt. Zum einen wird benötigt, wie groß das Dateisystem werden soll. Zum anderen wird benötigt, wie groß ein Block sein soll. Zuletzt wird auch benötigt, wie viele Blöcke es im Allgemeinen gibt. Als Übergabeparameter reichen dabei zwei der Werte, da der dritte Berechnet werden kann.

```
INodeSystem *INodeSystem::createSystem(size_t diskSize, size_t blockSize) {
    // Überprüfen, ob die Blockgröße nicht null ist, um eine Division durch null zu vermeiden
    if (blockSize == 0) {
        qCritical() << "Fehler: Die Blockgröße darf nicht null sein.";
    } else {
        auto *iNodeSystem = new INodeSystem();
        if (iNodeSystem == nullptr) {
            qCritical() << "Fehler: Speicher konnte nicht allokiert werden.";
        } else {
            // Ermittlung der Blockanzahl anhand der Festplatten- und Blockgröße
            size_t blockCount = (diskSize + blockSize - 1) / blockSize;
            iNodeSystem->data_blocks = new DataBlock[blockCount];
            if (iNodeSystem->data_blocks == nullptr) {
                free(iNodeSystem); // INodeSystem-Struktur freigeben, wenn die Allokation fehlschlägt.
                qCritical() << "Fehler: Speicher für das DatenBlock-Array konnte nicht allokiert werden.";
            } else {
                iNodeSystem->diskSize = diskSize;
                iNodeSystem->blockSize = blockSize;
                iNodeSystem->blockCount = blockCount;
                iNodeSystem->iNodes.push_back(INode(iNodeSystem->nextINodeNumber++, INodeType::DIRECTORY));

                // Initialisieren aller Blöcke als frei
                for (size_t i = 0; i < blockCount; i++) {
                    iNodeSystem->data_blocks[i] = DataBlock(BlockState::FREE);
                }

                return iNodeSystem;
            }
        }
    }
    return nullptr;
}
```

Abbildung 32: INode-System erstellen

Auf der Abbildung ist der Prozess der Erstellung des INode-Systems zu erkennen. Zuerst werden Validierungen durchgeführt, um ein Fehlverhalten zu verhindern. Anschließend wird die Anzahl der Datenblöcke errechnet, die in diesem System zur Verfügung stehen. Mit dieser Anzahl wird ein Datenblock-Array erstellt, welches die zur Verfügung stehenden Datenblöcke des Systems widerspiegelt.

Als nächstes folgt das Setzen der Attribute in dem System, sowie das Erstellen eines root-Ordners, die Initialisierung der Datenblöcke auf Frei und das Zurückgeben des fertig initialisierten Systems.

Der root-Ordner ist der oberste Ordner des Systems, in welchem alle weiteren Dateien und Ordner angelegt werden.

### 5.3 INode Datei anlegen

Die Erstellung einer Datei im INode-Dateisystem besteht im Wesentlichen aus 3 Schritten. Zum einen muss eine Validierung durchgeführt werden. Des Weiteren muss die Datei angelegt und die Datenblöcke reserviert werden.

```
int blocks_needed = (szFile + inode_system->blockSize - 1) / inode_system->blockSize;
if (inode_system->getFreeBlockCount() < blocks_needed) {
    qWarning() << "Fehler: Nicht genügend freier Speicherplatz verfügbar.";
    return false;
}

if(inode_system->getInodeFileFromPath(path) != nullptr){
    qWarning() << "Fehler: Datei existiert bereits.";
    return false;
}

if(inode_system->getInodeDirFromPath(path) == nullptr){
    createFolder(PathHelper::getDirectoryPath(path, '/'));
}

int inodeNumber = inode_system->nextINodeNumber++;
INode inode(inodeNumber, INodeType::FILE, fileend, filekind, owner, permissions,
            szFile, modification_time, creation_time);
DataHandler::fillDataBlocks(&inode, inode_system);
inode_system->iNodes.push_back(inode);
inode_system->addFileToDirectory(path, inodeNumber);
inode_system->setNewSizeToFolder(path, szFile);

return true;
```

Abbildung 33: INode-Datei anlegen

Wie im Bildausschnitt der Funktion *createFile(...)* zu erkennen ist, werden zuerst Validierungen bezüglich der Speicherplatzverfügbarkeit und dem Vorhandensein der anzulegenden Datei durchgeführt.

Falls entweder zu wenig Speicherplatz zur Verfügung steht oder die Datei bereits existiert, wird der Vorgang abgerochen und die jeweilige Fehlermeldung ausgegeben. Anderenfalls wird verprobt, ob es den Ordner, in welchem die Datei liegen soll, schon existiert, oder noch angelegt werden muss. Ist dieser Prozess durchgelaufen so wird ein neuer INode-File erstellt, welcher alle Informationen im Konstruktor übergeben bekommt.

Anschließend werden noch die Datenblöcke befüllt und die fertig erstellte und befüllte Datei dem richtigen Ordner zugeordnet und den Elternordnern die zusätzliche Größe mitgeteilt.

### 5.3.1 Datenblockspeicherung

```
void DataHandler::fillDataBlocks(INode *inode, INodeSystem *inode_system) {
    const size_t countBlocks = (inode->metadata.get_size() + inode_system->blockSize - 1) / inode_system->blockSize;
    size_t blocksLeft = countBlocks;
    srand((unsigned int) time(nullptr));
    for (int i = 0; i < 12 && blocksLeft > 0; i++) {
        int random = getRandomNumber(inode_system);

        DataBlock data_block;
        data_block.set_block_state(BlockState::OCCUPIED);
        inode_system->data_blocks[random] = data_block;
        inode->data[i] = random;

        blocksLeft--;
    }

    if (blocksLeft > 0) {
        inode->singleindirect = new Singleindirect;
        blocksLeft = fillVector(&inode->singleindirect->dataBlocks, blocksLeft, inode_system);
    }

    if (blocksLeft > 0) {
        inode->double_indirect = new DoubleIndirect;
        blocksLeft = fillVector1(inode->double_indirect->singleindirects, blocksLeft, inode_system);
    }

    if (blocksLeft > 0) {
        inode->triple_indirect = new TripleIndirect;
        fillVector2(inode->triple_indirect->double_indirects, blocksLeft, inode_system);
    }
}
```

Abbildung 34: Datenblockspeicherung von INodes

Beim Speichern der Daten einer INode wird wie folgt vorgegangen:

Wie in der Abbildung zu erkennen ist, werden zuerst die 12 direkten Pointer gefüllt. Wenn der Dateiinhalt noch nicht vollständig durch die 12 Pointer gespeichert wurde, wird der restliche Inhalt in den indirekten Blöcken gespeichert.

```
size_t DataHandler::fillVector(std::vector<int> *data_blocks, size_t blocks_left, INodeSystem *inode_system) {
    //Führe das Füllen solange aus, wie die übrige Blockzahl > 0 und die Größe der Liste
    //der Datenblock-Pointer nicht der Größe der Blockgröße/Größe eines Pointers
    //überschreitet.
    while (blocks_left > 0 && data_blocks->size() < inode_system->blockSize / sizeof(int)) {
        int random = getRandomNumber(inode_system);

        data_blocks->push_back(random);
        inode_system->data_blocks[random] = DataBlock(BlockState::OCCUPIED);
        blocks_left--;
    }
    return blocks_left;
}

size_t DataHandler::fillVector(std::vector<Singleindirect> singleindirects, size_t blocks_left,
    INodeSystem *inode_system) {
    //Führe das Füllen solange aus, wie die übrige Blockzahl > 0 und die Größe der Liste
    //der Einfach-Indirekten Pointer nicht der Größe der Blockgröße/Größe eines Pointers
    //überschreitet.
    while (blocks_left > 0 && singleindirects.size() < inode_system->blockSize / sizeof(int)) {
        Singleindirect singleindirect;
        blocks_left = fillVector(&singleindirect.dataBlocks, blocks_left, inode_system);
        singleindirects.push_back(singleindirect);
    }
    return blocks_left;
}

size_t DataHandler::fillVector(std::vector<DoubleIndirect> double_indirects, size_t blocks_left,
    INodeSystem *inode_system) {
    //Führe das Füllen solange aus, wie die übrige Blockzahl > 0 und die Größe der Liste
    //der Zweifach-Indirekten Pointer nicht der Größe der Blockgröße/Größe eines Pointers
    //überschreitet.
    while (blocks_left > 0 && double_indirects.size() < inode_system->blockSize / sizeof(int)) {
        DoubleIndirect double_indirect;
        blocks_left = fillVector(double_indirect.singleindirects, blocks_left, inode_system);
        double_indirects.push_back(double_indirect);
    }
    return blocks_left;
}
```

Abbildung 35: Datenblockspeicherung der indirekten Blöcke



In der obigen Abbildung sind die Methoden zur Verfüllung der indirekten Pointer zu erkennen. Die erste Methode stellt die Verfüllung des einfach-indirekten Pointers dar. Wie bei den 12 direkten Pointern werden die Datenblöcke zufällig belegt.

Die zweite Methode stellt die Verfüllung des zweifach-indirekten Pointers dar, die dritte wiederum für den dreifach-indirekten Pointer.

### 5.3.2 Zufällige Belegung

Die Datenblöcke der 12 direkten, wie auch aller indirekten Pointer werden zufällig belegt, damit eine Defragmentierung Wirkung zeigen kann.

```
int DataHandler::getRandomNumber(INodeSystem* inode_system){
    int random;
    int noPlaceFound = 0;
    do {
        random = arc4random() % inode_system->blockCount;
        noPlaceFound++;
    } while (inode_system->data_blocks[random].get_block_state() != BlockState::FREE && noPlaceFound < 10);

    //wenn nach 10-maligen Suchen einer Zufallszahl kein Treffer erzielt wurde, wird die erste freie Stelle
    //im Blockspeicher genommen, um lange Wartezeiten zu verhindern.
    if(noPlaceFound >= 10){
        random = -1;
        for (int var = 0; var < inode_system->blockCount; ++var) {
            if(inode_system->data_blocks[var].get_block_state() == BlockState::FREE){
                random = var;
                break;
            }
        }
    }
    return random;
}
```

Abbildung 36: Bereitstellung eines freien Datenblocks

Die zufällige Belegung eines Blocks, wird durch die oben gezeigte Abbildung bereitgestellt. Es wird versucht einen freien Datenblock zu finden. Wenn nach 10-maligen Versuchen kein freier Block gefunden wurde, wird die Suche abgebrochen und der erste freie Datenblock aus dem Datenblockspeicher ermittelt und die Referenz zurückgegeben. Dies hat den Hintergrund lange Wartezeiten zu verhindern falls nur noch wenige Blöcke zur Verfügung stehen. Falls vorher ein freier Block gefunden wurde, wird die Referenz zurückgegeben.

## 5.4 INode-Ordner anlegen

Das Anlegen eines Ordners einer INode funktioniert ähnlich wie bei einer Datei.

```
bool INodeAdapter::createFolder(std::string path) {
    //besorgt sich anhand des Pfades einen Order, da ein noch nicht existierender Ordner ein Null liefert
    INode *latestDir = inode_system->getInodeDirFromPath(path);
    if (latestDir == nullptr) {
        std::vector<std::string> result = PathHelper::split(path, '/');
        latestDir = inode_system->getRootDir();
        for (const auto &part: result) {
            //verprobt auf vorhandensein des aktuellen ordners
            int iNodeNumber = inode_system->getIndexByName(part, latestDir->directoryEntries);
            if (iNodeNumber == -1) {
                int newInodeNumber = inode_system->nextInodeNumber++;
                latestDir->directoryEntries[part] = newInodeNumber;
                INode dir(newInodeNumber, INodeType::DIRECTORY);
                inode_system->iNodes.push_back(dir);
                latestDir = &inode_system->iNodes.back();
            } else {
                latestDir = inode_system->getInodeByIndex(iNodeNumber);
            }
        }
    } else {
        qWarning() << "Ordner existiert bereits.";
        return false;
    }
    return true;
}
```

Abbildung 37: INode-Ordner anlegen

Zu Beginn wird verprobt, ob die anzulegende Ordnerstruktur bereits existiert, falls dem so ist, wird eine Fehlermeldung ausgegeben und die Operation abgebrochen. Andernfalls wird der Pfad an dem Slash aufgespalten, um die einzelnen Ordner zu extrahieren. Anschließend werden diese durchgegangen und auf Vorhandensein verprobt. Existiert der Ordner, so wird dieser anhand der INode-Nummer ermittelt, andernfalls wird dieser angelegt, der Liste aller INodes hinzugefügt und dem aktuellen Ordner mitgeteilt, dass dieser nun einen weiteren Ordner beinhaltet.

## 5.5 INode-Dateien und Ordner löschen

Um eine Datei oder ein Ordner aus dem INode Dateisystem zu löschen, müssen die oben genannten Schritte (Abschnitt 5.3 & 5.4) rückgängig gemacht werden.

### 5.5.1 INode-Dateien löschen

Die Methode *removeFile(...)* kann sowohl für Dateien, als auch Ordner angewandt werden.

```
bool INodeAdapter::removeFile(std::string path) {
    vector<string> split = PathHelper::split(path, '/');
    string fileName = split.back();
    int index = inode_system->getIndexByName(fileName);
    INode *dir;
    if(inode_system->isFile(index)){
        dir = inode_system->getInodeDirFromPath(path);
    }else{
        split.pop_back();
        dir = inode_system->getInodeDirFromPath(PathHelper::join(split, '/'));
    }

    if(dir == nullptr){
        qWarning() << "Datei mit dem Pfad: " << path << " nicht gefunden.";
        return false;
    }

    //Die Verzweigung der Datei wird aus dem Oberordner entnommen
    dir->directoryEntries.erase(fileName);

    auto it = std::find_if(inode_system->iNodes.begin(), inode_system->iNodes.end(), [index](const INode &node) {
        return node.get_index() == index;
    });

    if (it != inode_system->iNodes.end()) {
        if(it->isFile()){
            vector<int> blockNumbers = it->getAllUsedBlockNumbers();
            for (int i: blockNumbers) {
                inode_system->data_blocks[i].clearData();
            }
            inode_system->setNewSizeToFolder(path, -it->get_metadata().get_size());
        }
        inode_system->iNodes.erase(it);
        return true;
    }
    return false;
}
```

Abbildung 38: INode-Datei löschen

Zu Beginn wird der Elementname und der Index des zu löschenden Elements ermittelt. Anschließend wird je nachdem, ob es sich bei dem Element um eine Datei oder einen Ordner handelt, der Elternordner ermittelt. Anschließend wird die Verzweigung des Namens aus dem Elternordner entnommen. Des Weiteren wird anhand des Indizes die Datei als Iterator bereitgestellt und auf Vorhandensein verprobt.

Nun wird abgefragt, ob es sich um eine Datei handelt. Ist dies der Fall, so werden alle der Datei angehörigen Datenblöcke ermittelt und anschließend wieder als frei definiert. Im folgenden Schritt wird den Elternordnern die neue Größe mitgeteilt. Zum Schluss wird die Datei aus dem INode-Speicher entfernt.

### 5.5.2 INode-Ordner löschen

Beim Löschen eines Ordners müssen alle darin liegenden Ordner und Dateien gelöscht werden. Wobei Ordner rekursiv gelöscht werden müssen.

```
bool INodeAdapter::removeFolder(string path) {
    vector<string> filesToRemove;

    INode* dir = inode_system->getNodeDirFromPath(path);
    vector<string> split = PathHelper::split(path, '/');
    string dirName = split.back();
    split.pop_back();
    string newPath = PathHelper::join(split, '/');
    filesToRemove.push_back(path);
    inode_system->getDirectoryTree(&filesToRemove, newPath + "/", dir);
    std::reverse(filesToRemove.begin(), filesToRemove.end());

    for(string &file : filesToRemove){
        if(!removeFile(file)){
            return false;
        }
    }
    INode *dir_to_delete = inode_system->getNodeDirFromPath(newPath);
    if(dir_to_delete == nullptr){
        qCritical() << "INode konnte nicht gefunden werden.";
        return false;
    }
    return true;
}
```

Abbildung 39: INode-Ordner löschen

Zu Beginn wird der zu entfernende Ordner, anschließend dessen Namen und der Dateipfad des Elternordners ermittelt, um anhand dieser Attribute den vollständigen Verzeichnisbaum zu bekommen. Dieser Verzeichnisbaum wird nun invertiert, um am untersten Punkt anzufangen zu löschen.

Anschließend wird der Pfad des zu entfernenden Ordners der Liste an zu löschenden Elementen hinzugefügt und durch diese iteriert. Infolgedessen wird nach und nach immer das dementsprechende Element gelöscht