

## Aufgabenblatt 9 – Softwaretechnik

### A1 Git-Lebenszyklus von Dateien (State)

#### [a] UML-Klassendiagramm

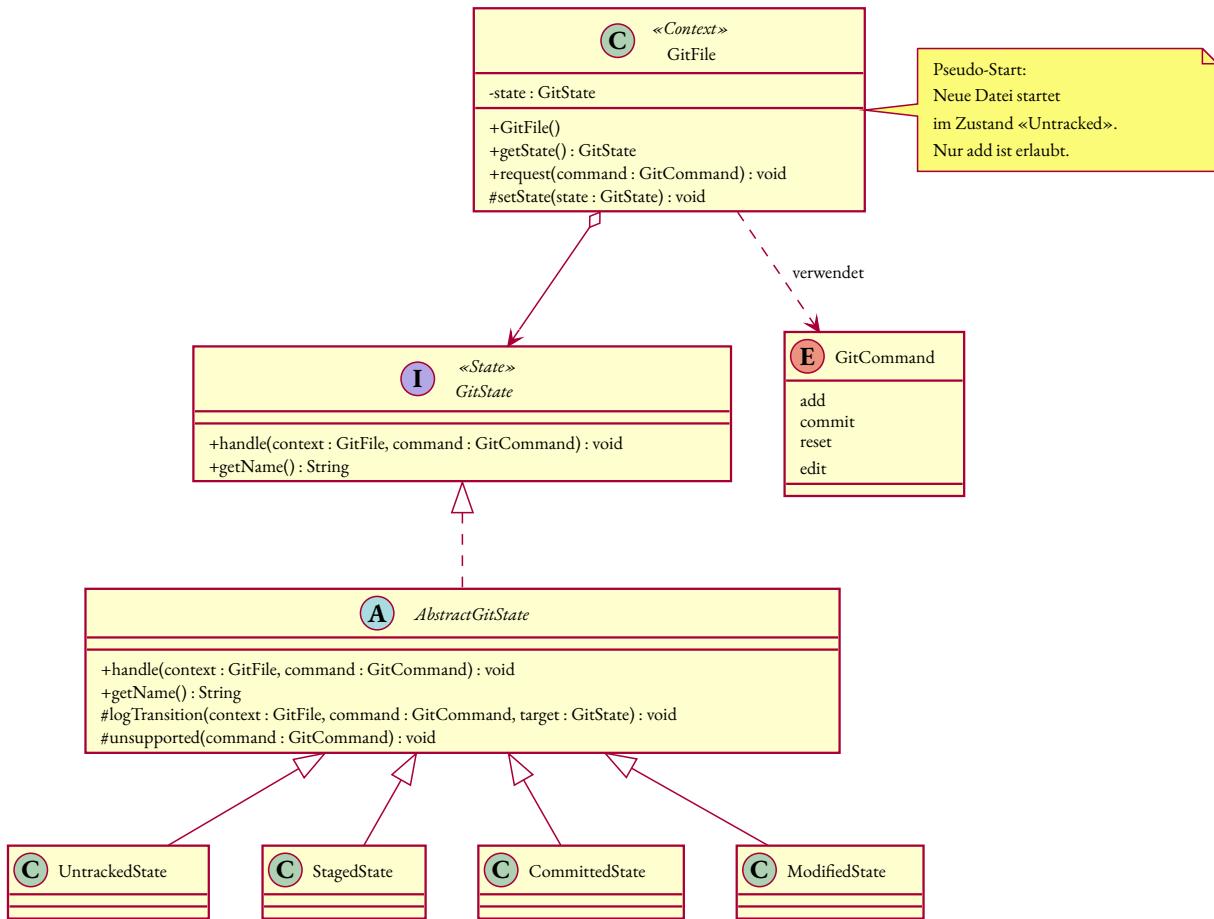


Abbildung 1: Klassendiagramm State-Entwurfsmuster für den Git-Lebenszyklus

#### [b] Java-Implementierung der Zustände

```
public enum GitCommand {
    ADD, COMMIT, RESET, EDIT
}

public interface GitState {

    /**
     * Behandelt den angegebenen Befehl für die Datei im aktuellen Zustand.
     *
     * @param file Kontextobjekt (Context)
    
```

```
* @param command auszuführender Git-Befehl
* @throws UnsupportedOperationException bei nicht erlaubtem Übergang
*/
void handle(GitFile file, GitCommand command);

/**
 * Name des Zustands (nur für Logging / Debugging).
 */
String getName();
}

/**
 * Context im State-Pattern.
 */
public class GitFile {

    private GitState state;

    public GitFile() {
        // Pseudo-Startzustand: Datei ist zunächst untracked.
        this.state = new UntrackedState();
    }

    void setState(GitState state) {
        this.state = state;
    }

    public GitState getState() {
        return state;
    }

    /**
     * Öffentliche Schnittstelle für den Client. Delegiert an das aktuelle State-Objekt.
     */
    public void request(GitCommand command) {
        state.handle(this, command);
    }
}

/**
 * Gemeinsame Basisklasse für konkrete Zustände.
 * Kapselt Logging und Fehlerbehandlung.
 */
abstract class AbstractGitState implements GitState {

    protected void logTransition(GitFile file,
                                GitCommand command,
```

```
GitState targetState) {  
  
    System.out.printf("Command %-6s: %s -> %s%n",
                       command, getName(), targetState.getName());
    file.setState(targetState);
}  
  
protected void unsupported(GitCommand command) {
    throw new UnsupportedOperationException(
        "Command " + command + " ist im Zustand " + getName() + " nicht erlaubt.");
}  
}  
  
class UntrackedState extends AbstractGitState {  
  
    @Override
    public void handle(GitFile file, GitCommand command) {
        switch (command) {
            case ADD -> logTransition(file, command, new StagedState());
            default -> unsupported(command);
        }
    }
}  
  
    @Override
    public String getName() {
        return "Untracked";
    }
}  
  
class StagedState extends AbstractGitState {  
  
    @Override
    public void handle(GitFile file, GitCommand command) {
        switch (command) {
            case COMMIT -> logTransition(file, command, new CommittedState());
            case RESET   -> logTransition(file, command, new UntrackedState());
            default       -> unsupported(command);
        }
    }
}  
  
    @Override
    public String getName() {
        return "Staged";
    }
}  
  
class CommittedState extends AbstractGitState {
```

```
@Override
public void handle(GitFile file, GitCommand command) {
    switch (command) {
        case EDIT -> logTransition(file, command, new ModifiedState());
        default     -> unsupported(command);
    }
}

@Override
public String getName() {
    return "Committed";
}
}

class ModifiedState extends AbstractGitState {

    @Override
    public void handle(GitFile file, GitCommand command) {
        switch (command) {
            case ADD   -> logTransition(file, command, new StagedState());
            case RESET -> logTransition(file, command, new CommittedState());
            default      -> unsupported(command);
        }
    }
}

@Override
public String getName() {
    return "Modified";
}
}
```

### [c] Konsolenanwendung zur Ausführung der Befehlskette

```
public class GitLifecycleDemo {

    public static void main(String[] args) {
        GitFile file = new GitFile();

        GitCommand[] script = {
            GitCommand.ADD,
            GitCommand.COMMIT,
            GitCommand.EDIT,
            GitCommand.ADD,
            GitCommand.RESET
        };

        for (GitCommand command : script) {
```

```

try {
    file.request(command);
} catch (UnsupportedOperationException ex) {
    System.out.println("Fehler: " + ex.getMessage());
    break;
}
}

System.out.println("Endzustand: " + file.getState().getName());
}
}

```

Die tatsächlichen Git-Befehle werden nicht ausgeführt, es wird lediglich der Zustandsübergang geloggt und bei verbotenen Übergängen eine UnsupportedOperationException geworfen.

## A2 Dateimanagement (Command & Memento)

### [a] UML-Klassendiagramm

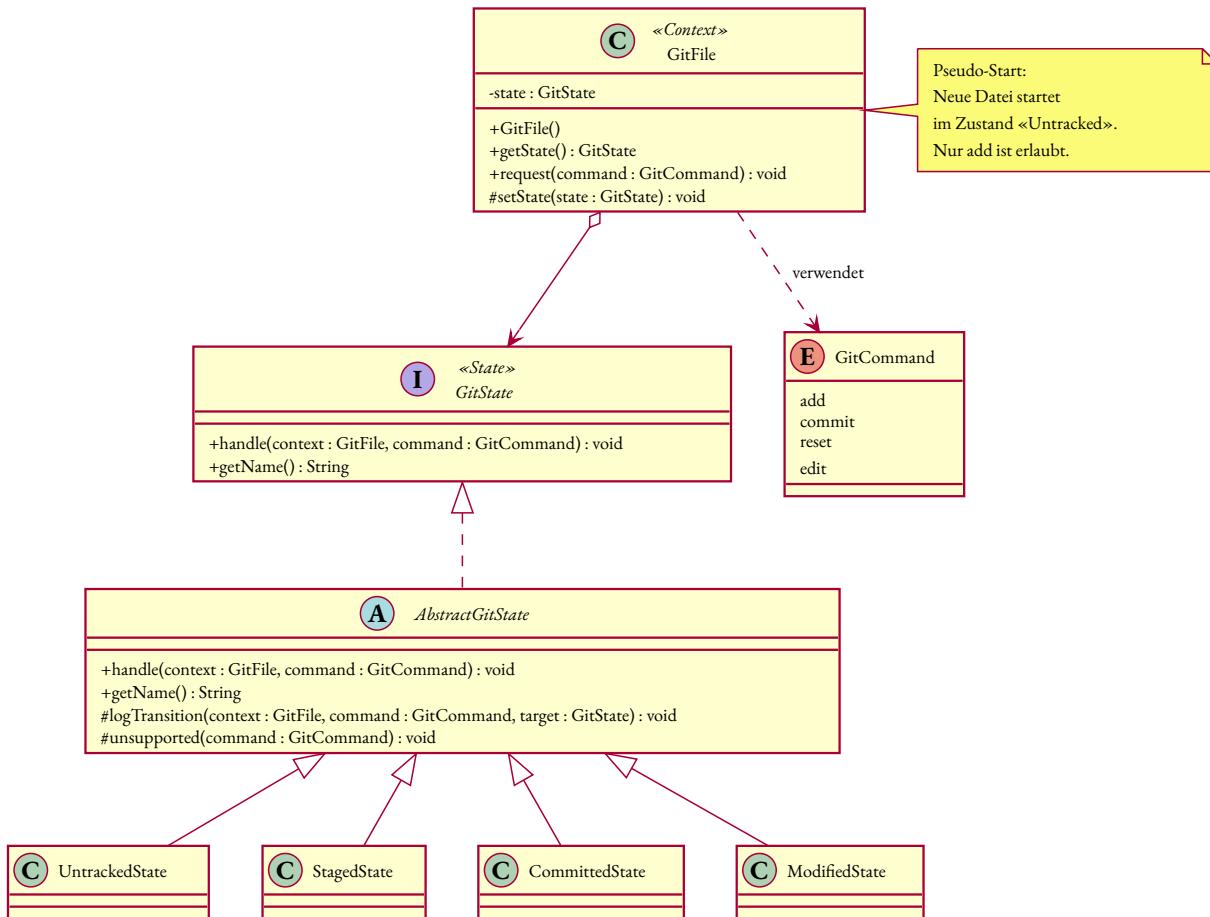


Abbildung 2: Klassendiagramm Dateimanagement mit Command & Memento

**[b] Java-Implementierung**

```
import java.time.LocalDateTime;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Objects;

/**
 * Originator: repräsentiert eine Datei.
 */
public final class File {

    private String name;
    private String path;
    private final LocalDateTime created;
    private boolean readOnly;

    public File(String name, String path) {
        this.name = Objects.requireNonNull(name);
        this.path = Objects.requireNonNull(path);
        this.created = LocalDateTime.now();
        this.readOnly = false;
    }

    public String getName() {
        return name;
    }

    public String getPath() {
        return path;
    }

    public LocalDateTime getCreated() {
        return created;
    }

    public boolean isReadOnly() {
        return readOnly;
    }

    public void setName(String name) {
        this.name = Objects.requireNonNull(name);
    }

    public void setPath(String path) {
        this.path = Objects.requireNonNull(path);
    }
}
```

```
public void setReadOnly(boolean readOnly) {
    this.readOnly = readOnly;
}

Backup createBackup() {
    return new Backup(name, path, created, readOnly);
}

void restoreBackup(Backup backup) {
    this.name = backup.getName();
    this.path = backup.getPath();
    this.readOnly = backup.isReadOnly();
}

@Override
public String toString() {
    return "File{" +
        "name='" + name + '\'' +
        ", path='" + path + '\'' +
        ", created=" + created +
        ", readOnly=" + readOnly +
        '}';
}

/**
 * Memento: unveränderliches Backup-Objekt.
 */
final class Backup {

    private final String name;
    private final String path;
    private final LocalDateTime created;
    private final boolean readOnly;

    Backup(String name, String path, LocalDateTime created, boolean readOnly) {
        this.name = name;
        this.path = path;
        this.created = created;
        this.readOnly = readOnly;
    }

    String getName() {
        return name;
    }

    String getPath() {
```

```
        return path;
    }

    LocalDateTime getCreated() {
        return created;
    }

    boolean isReadOnly() {
        return readOnly;
    }
}

/**
 * Command + Caretaker.
 */
public abstract class FileCommand {

    private final File file;
    private Backup backup;

    protected FileCommand(File file) {
        this.file = Objects.requireNonNull(file);
    }

    public final void execute() {
        backup = file.createBackup();
        doExecute();
    }

    public final void restore() {
        if (backup == null) {
            throw new IllegalStateException("Kein Backup vorhanden.");
        }
        file.restoreBackup(backup);
    }

    protected File getFile() {
        return file;
    }

    protected abstract void doExecute();
}

/**
 * ConcreteCommand: Umbenennen.
 */
public final class RenameCommand extends FileCommand {
```

```
private final String newName;

public RenameCommand(File file, String newName) {
    super(file);
    this.newName = Objects.requireNonNull(newName);
}

@Override
protected void doExecute() {
    getFile().setName(newName);
}
}

/**
 * ConcreteCommand: Verschieben.
 */
public final class MoveCommand extends FileCommand {

    private final String newPath;

    public MoveCommand(File file, String newPath) {
        super(file);
        this.newPath = Objects.requireNonNull(newPath);
    }

    @Override
    protected void doExecute() {
        getFile().setPath(newPath);
    }
}

/**
 * ConcreteCommand: Schreibschutz setzen/entfernen.
 */
public final class ProtectCommand extends FileCommand {

    private final boolean readOnly;

    public ProtectCommand(File file, boolean readOnly) {
        super(file);
        this.readOnly = readOnly;
    }

    @Override
    protected void doExecute() {
        getFile().setReadOnly(readOnly);
    }
}
```

```
    }

}

/**  
 * Invoker: verwaltet Undo-/Redo-Stacks.  
 */  
public final class FileManager {  
  
    private final Deque<FileCommand> undoHistory = new ArrayDeque<>();  
    private final Deque<FileCommand> redoHistory = new ArrayDeque<>();  
  
    public void invoke(FileCommand command) {  
        command.execute();  
        undoHistory.push(command);  
        redoHistory.clear();  
    }  
  
    public void undo() {  
        if (undoHistory.isEmpty()) {  
            System.out.println("Nichts zum Rückgängig machen.");  
            return;  
        }  
        FileCommand command = undoHistory.pop();  
        command.restore();  
        redoHistory.push(command);  
    }  
  
    public void redo() {  
        if (redoHistory.isEmpty()) {  
            System.out.println("Nichts zum Wiederholen.");  
            return;  
        }  
        FileCommand command = redoHistory.pop();  
        command.execute();  
        undoHistory.push(command);  
    }  
}
```

### [c] Konsolenanwendung für die geforderte Befehlskette

```
public class FileManagerDemo {  
  
    public static void main(String[] args) {  
        File file = new File("MyFile.txt", "/tmp");  
        FileManager manager = new FileManager();  
  
        printState("Neue Datei angelegt", file);
```

```
// Datei verschieben
manager.invoke(new MoveCommand(file, "/home/user/documents"));
printState("Nach Move", file);

// Datei als schreibgeschützt setzen
manager.invoke(new ProtectCommand(file, true));
printState("Nach Protect", file);

// Letzten Befehl rückgängig machen
manager.undo();
printState("Nach Undo (Protect)", file);

// Letzten Befehl rückgängig machen
manager.undo();
printState("Nach Undo (Move)", file);

// Letzten Befehl wiederholen
manager.redo();
printState("Nach Redo (Move)", file);
}

private static void printState(String label, File file) {
    System.out.println(label + ": " + file);
}
}
```

Die tatsächliche Interaktion mit dem Dateisystem findet nicht statt; es werden lediglich die Attribute des File-Objekts angepasst und in der Konsole ausgegeben.