# JDK-Migrations-Leitfaden

## Elyndras bewährte 4-Phasen-Strategie: JDK 8 → 17 ohne Chaos

*Von Elyndra Valen, Senior Entwicklerin & Code-Archäologin*
*Java Fleet Systems Consulting, Essen-Rüttenscheid*

## 🏺 Vorwort: Meine Code-Archäologie-Erfahrung

Liebe Java-Entwickler-Gilde!

Nach drei Legacy-Modernisierungen (Legacy Modernizations) und der erfolgreichen JDK 8→17 Migration eines 10-Jahre-alten Enterprise-Systems, teile ich hier meine **erprobte 4-Phasen-Strategie** mit dir.

**Dieses Handbuch (Playbook) ist entstanden aus:**

- ✅ **1 Horror-Migration** (die fast gescheitert wäre)
- ✅ **2 erfolgreiche Migrationen** (mit der hier beschriebenen Strategie)
- ✅ **47 Community-Horror-Stories** (von euren E-Mails!)
- ✅ **Hunderte Stunden** Code-Archäologie-Arbeit

**Das Ziel:** Dir die **Schmerzen und Fehler** zu ersparen, die ich gemacht habe, und eine **systematische, risikoarme** Herangehensweise zu geben.

### 🎯 Für wen ist dieses Handbuch?

**Perfekt für:**

- **Senior Developers** mit Legacy-Code-Verantwortung
- **Tech Leads** die JDK-Migrations-Projekte planen
- **Teams** die von JDK 8/11 auf 17+ migrieren wollen
- **Alle** die Angst vor Breaking Changes haben

**Vorausgesetztes Wissen:**

- **Java-Grundlagen** (Generics, Collections, etc.)
- **Build-Tools** (Maven oder Gradle)
- **Grundlagen Testing** (JUnit, Mockito)
- **Basis Git-Knowledge**

# 📚 Handbuch-Struktur (Playbook Structure)

**Phase 0:** Mindset & Bewertung (Assessment) (1 Woche)
**Phase 1:** Characterization Tests (2-3 Wochen)
**Phase 2:** Dependency-Detox (1-2 Wochen)
**Phase 3:** Code-Refactoring (2-4 Wochen)
**Phase 4:** JDK-Migration (1-2 Wochen)
**Phase 5:** Modernization (2-4 Wochen)

**Gesamt-Timeline:** 8-16 Wochen (je nach System-Größe)

# 🧠 Phase 0: Mindset & Bewertung (Assessment)

## 🎯 Der wichtigste Mindset-Shift

**STOP thinking:** *"This legacy code is crap, let's rewrite everything!"*
**START thinking:** *"This code tells a story - let's understand it first."*

**Legacy-Code-Respekt-Prinzipien (Legacy Code Respect Principles):**

1. **Jede Codezeile (Line of Code) hatte einen Grund** (auch wenn er 2014 dokumentiert wurde)
2. **Merkwürdige Geschäftsregeln (Weird Business Rules)** sind oft Bug-Fixes von längst vergessenen Problemen
3. **"Temporary Hacks"** sind manchmal **permanente Geschäftsanforderungen (Business Requirements)**
4. **Performance-"Optimizations"** können **kritische Produktions-Workarounds (Critical Production Workarounds)** sein

## 📊 System-Bewertung (System Assessment) (Tag 1-2) - DETAILLIERT

Die Bewertung (Assessment) ist der **wichtigste Teil** der ganzen Migration! Hier entscheidet sich, ob du 8 Wochen oder 8 Monate brauchst.

**Schritt 1: Basis-Statistiken sammeln - Code-Inventur (Code Inventory)**

**1.1 Codezeilen-Analyse (Lines of Code Analysis):**

**🐧 Linux/macOS (Bash):**

```
# Gesamte Codezeilen (Total Lines of Code)
find . -name "*.java" -not -path "*/target/*" -not -path "*/test/*" | xargs wc -
l | tail -1

# Codezeilen pro Package (Top 10 größte) - Lines of Code per Package
find . -name "*.java" -not -path "*/target/*" | \
  sed 's|.*src/main/java/||' | sed 's|/[^/]*\.java$||' | sort | uniq -c | sort -
rn | head -10
```

### 🪟 Windows (PowerShell):

```
# Gesamte Codezeilen (Total Lines of Code)
Get-ChildItem -Recurse -Include "*.java" -Exclude "*target*","*test*" |
  Get-Content | Measure-Object -Line | Select-Object -ExpandProperty Lines

# Codezeilen pro Package (Top 10 größte)
Get-ChildItem -Recurse -Include "*.java" -Exclude "*target*" |
  ForEach-Object {
    $package = $_.DirectoryName -replace ".*src\\main\\java\\", "" -replace
"\\", "."
    [PSCustomObject]@{ Package = $package; File = $_.Name }
  } | Group-Object Package |
  Sort-Object Count -Descending | Select-Object -First 10 |
  Format-Table Name, Count
```

### Beispiel-Ausgabe und Erklärung:

```
Lines of Code: 234567 total
= ~235k LOC = Medium-Projekt (12-16 Wochen Migration)

Top Packages:
com.company.payment.processor 1247 files
com.company.user.management 891 files
com.company.reporting.engine 634 files
= payment/processor ist der größte Bereich (focus area!)
```

### Was bedeutet das?

- **<50k LOC**: Kleines Projekt, 6-8 Wochen Migration
- **50-200k LOC**: Mittleres Projekt, 8-12 Wochen Migration
- **>200k LOC**: Großes Projekt, 12+ Wochen Migration
- Die größten Packages sind deine **Risikobereiche** - hier musst du besonders vorsichtig sein!

### 1.2 Komplexitäts-Hotspots finden (Complexity Hotspots):

### 🐧 Linux/macOS:

```
# Komplexe Dateien identifizieren (>500 LOC)
find . -name "*.java" -not -path "*/target/*" -exec wc -l {} + | \
  awk '$1 > 500 {print $1 " " $2}' | sort -rn

# Methodenkomplexität schätzen (if/for/while statements)
grep -r "if\|for\|while\|switch" --include="*.java" src/main/java/ | \
  cut -d: -f1 | sort | uniq -c | sort -rn | head -10
```

### 🪟 Windows (PowerShell):

```
# Komplexe Dateien identifizieren (>500 LOC)
Get-ChildItem -Recurse -Include "*.java" -Exclude "*target*" |
  ForEach-Object {
    $lineCount = (Get-Content $_.FullName | Measure-Object -Line).Lines
    if ($lineCount -gt 500) {
      [PSCustomObject]@{ Lines = $lineCount; File = $_.FullName }
    }
  } | Sort-Object Lines -Descending | Format-Table

# Methodenkomplexität schätzen
Get-ChildItem -Recurse -Include "*.java" -Path "src/main/java" |
  ForEach-Object {
    $complexity = (Select-String -Path $_.FullName -Pattern "if |for |while |
switch ").Count
```

```
    [PSCustomObject]@{ File = $_.Name; Complexity = $complexity }
  } | Sort-Object Complexity -Descending | Select-Object -First 10
```

**Beispiel-Ausgabe und Erklärung:**

```
1247 lines: ./src/main/java/com/company/PaymentProcessor.java <-- MONSTER!
 891 lines: ./src/main/java/com/company/UserManager.java
 634 lines: ./src/main/java/com/company/ReportGenerator.java

PaymentProcessor.java: 89 complexity statements <-- complexity nightmare
UserManager.java: 45 complexity statements
```

**Was bedeutet das?**

- **>500 Zeilen**: Monster-Dateien, die schwer zu migrieren sind
- **>30 Komplexitäts-Statements**: Sehr komplexe Methoden, hohes Risiko für Bugs bei Migration
- **PaymentProcessor.java** ist dein **größtes Risiko** - hier brauchst du die besten Tests!

**1.3 Git-History-Analyse (Change-Frequency):**

🐧 **Linux/macOS:**

```
# Dateien die am häufigsten geändert werden (hotspots)
git log --name-only --pretty=format: --since="1 year ago" | \
  grep -v "^$" | sort | uniq -c | sort -rn | head -20

# Code-Churn-Rate (additions + deletions)
git log --numstat --since="6 months ago" --pretty=format: | \
  awk '{add+=$1; del+=$2} END {print "Code churn: " add+del " lines changed in 6
months"}'
```

🪟 **Windows (PowerShell):**

```
# Dateien die am häufigsten geändert werden (hotspots)
git log --name-only --pretty=format: --since="1 year ago" |
  Where-Object { $_ -ne "" } | Group-Object |
  Sort-Object Count -Descending | Select-Object -First 20 |
  Format-Table Name, Count

# Code-Churn-Rate berechnen
$churnData = git log --numstat --since="6 months ago" --pretty=format: |
  Where-Object { $_ -match "^\d+\s+\d+" } |
  ForEach-Object {
    $parts = $_ -split "\s+"
    [int]$parts[0] + [int]$parts[1]
  } | Measure-Object -Sum

Write-Host "Code churn: $($churnData.Sum) lines changed in 6 months"
```

**Beispiel-Ausgabe und Erklärung:**

```
Hot Files:
PaymentProcessor.java: 127 changes <-- oft geändert = high risk!
UserService.java: 89 changes
pom.xml: 67 changes

Code churn: 45678 lines changed in 6 months
= High churn rate = mehr Regression-Risk
```

**Was bedeutet das?**

- **Oft geänderte Dateien** haben hohes Regressionsrisiko
- **Hohe Churn-Rate** bedeutet aktive Entwicklung - mehr Konfliktpotential
- Diese Dateien brauchen die **besten Tests** vor der Migration!

### 1.4 Test-Coverage-Realitäts-Check (Test Coverage Reality Check):

#### 🐧 Linux/macOS:

```
# Vorhandene Test-Dateien zählen
find . -name "*Test.java" -o -name "*Tests.java" | wc -l

# Ratio Test-Code zu Production-Code
production_files=$(find . -name "*.java" -not -path "*/test/*" | wc -l)
test_files=$(find . -name "*.java" -path "*/test/*" | wc -l)
echo "Test Ratio: $test_files tests for $production_files production files"
```

#### 🪟 Windows (PowerShell):

```
# Vorhandene Test-Dateien zählen
$testFiles = Get-ChildItem -Recurse -Include "*Test.java","*Tests.java" |
Measure-Object
Write-Host "Test files found: $($testFiles.Count)"

# Ratio Test-Code zu Production-Code
$productionFiles = (Get-ChildItem -Recurse -Include "*.java" -Exclude "*test*" |
Measure-Object).Count
$testFiles = (Get-ChildItem -Recurse -Include "*.java" | Where-Object
{ $_.DirectoryName -like "*test*" } | Measure-Object).Count
$ratio = [math]::Round(($testFiles / $productionFiles) * 100, 1)
Write-Host "Test Ratio: $testFiles tests for $productionFiles production files
($ratio%)"
```

#### 🎯 Cross-Platform (Maven):

```
# JaCoCo Coverage (funktioniert auf allen Plattformen)
mvn clean test jacoco:report
# Report: target/site/jacoco/index.html
```

**Beispiel-Ausgabe und Erklärung:**

```
Test Ratio: 45 tests for 234 production files (19.2%)
= Low test coverage (needs improvement!)
```

**Was bedeutet das?**

- **<30% Test-Coverage**: Katastrophal - Migration sehr riskant
- **30-50%**: Schwach - du brauchst viel mehr Tests vor der Migration
- **50-70%**: Okay - einige zusätzliche Tests nötig
- **>70%**: Gut - Migration ist machbar

### Schritt 2: Abhängigkeits-Inventar (Dependency Inventory) - Dependency-Hell-Prognose

#### 2.1 Dependency-Tree-Analyse:

#### 🌍 Cross-Platform (Maven/Gradle):

```
# Maven (Windows + Linux)
mvn dependency:tree -Dverbose > dependency-analysis.txt

# Gradle (Windows + Linux)
./gradlew dependencies > dependency-analysis.txt

# PowerShell-friendly Maven
mvn dependency:tree -Dverbose | Out-File -FilePath "dependency-analysis.txt" -
Encoding UTF8
```

#### 🪟 Windows-specific Analysis:

```
# Dependency-Count-Analyse
$depTree = mvn dependency:tree -Dverbose | Out-String
$totalDeps = ($depTree | Select-String "\[INFO\].*:.*:.*:.*:.*").Count
$directDeps = ($depTree | Select-String "^\[INFO\] \+\-" -AllMatches).Count

Write-Host "Total dependencies (including transitive): $totalDeps"
Write-Host "Direct dependencies: $directDeps"
```

#### Was bedeutet das?

- **Transitive Dependencies**: Abhängigkeiten von deinen Abhängigkeiten - können versteckte Probleme verursachen
- **Direct Dependencies**: Die, die du direkt in pom.xml definiert hast
- **>50 Direct Dependencies**: Potentiell problematisch für Migration

#### 2.2 Veraltete Abhängigkeits-Analyse (Outdated Dependencies Analysis):

#### 🌍 Cross-Platform:

```
# Maven Versions Plugin (Windows + Linux)
mvn versions:display-dependency-updates > outdated-deps.txt

# Gradle Versions Plugin
./gradlew dependencyUpdates > outdated-deps.txt
```

#### 🪟 Windows PowerShell Analysis:

```
# Analysiere Maven Versions Output
mvn versions:display-dependency-updates | Out-String |
  Select-String "MAJOR|MINOR" -Context 2 |
  ForEach-Object { Write-Host $_.Line -ForegroundColor Yellow }

# Beispiel: Spring Framework Version Check
$pomContent = Get-Content "pom.xml" -Raw
if ($pomContent -match '<spring\.version>(.*?)</spring\.version>') {
  $currentSpringVersion = $matches[1]
  Write-Host "Current Spring Version: $currentSpringVersion" -ForegroundColor
Red
  if ($currentSpringVersion -lt "5.3.0") {
    Write-Host "WARNING: Spring 4.x does NOT support JDK 17!" -ForegroundColor
Red
  }
}
```

**Was bedeutet das?**

- **MAJOR Updates**: Große Versionssprünge, oft mit Breaking Changes
- **MINOR Updates**: Kleinere Updates, meist sicher
- **Spring 4.x**: Funktioniert NICHT mit JDK 17 - muss aktualisiert werden

**2.3 Sicherheits-Vulnerability-Scan:**

🌍 **Cross-Platform (Maven):**

```
# OWASP Dependency Check (dauert 10-15 Minuten!)
mvn org.owasp:dependency-check-maven:check

# Report analysieren
# Windows: start target/dependency-check-report.html
# macOS: open target/dependency-check-report.html
# Linux: xdg-open target/dependency-check-report.html
```

🪟 **Windows PowerShell Analysis:**

```
# Vulnerability Summary aus HTML-Report extrahieren
$reportPath = "target/dependency-check-report.html"
if (Test-Path $reportPath) {
  $reportContent = Get-Content $reportPath -Raw
  $criticalVulns = ($reportContent | Select-String "Critical" -
AllMatches).Matches.Count
  $highVulns = ($reportContent | Select-String "High" -AllMatches).Matches.Count

  Write-Host "Security Vulnerabilities Found:" -ForegroundColor Red
  Write-Host "  Critical: $criticalVulns" -ForegroundColor Red
  Write-Host "  High: $highVulns" -ForegroundColor Yellow

  # HTML-Report öffnen
  Start-Process $reportPath
} else {
  Write-Host "Run 'mvn org.owasp:dependency-check-maven:check' first" -
ForegroundColor Yellow
}
```

**Was bedeutet das?**

- **Critical Vulnerabilities**: Sofortige Sicherheitsrisiken - müssen vor Migration behoben werden
- **High Vulnerabilities**: Ernste Sicherheitsprobleme - sollten behoben werden
- **CVE (Common Vulnerabilities and Exposures)**: Standard für Sicherheitslücken-Dokumentation

**Schritt 3: JDK-Kompatibilitäts-Check (JDK Compatibility Check) - Der Wahrheitsmoment**

**3.1 jdeps-Analyse (Java Dependencies Analysis):**

🌍 **Cross-Platform Setup:**

```
# Erst das Projekt kompilieren (Windows + Linux)
mvn clean compile

# Oder Gradle
./gradlew compileJava
```

### 🐧 Linux/macOS jdeps:

```
# jdeps auf compiled classes laufen lassen
jdeps --check-modules --module-path . target/classes/ 2>&1 | tee jdeps-
report.txt

# Detaillierte Analyse pro Package
jdeps --print-module-deps target/classes/ 2>&1 | tee jdeps-modules.txt

# sun.* package usage (illegal in JDK 9+)
jdeps --jdk-internals target/classes/ 2>&1 | tee jdeps-internals.txt
```

### 🪟 Windows PowerShell jdeps:

```
# jdeps Analysis mit PowerShell
$jdepsOutput = jdeps --check-modules --module-path . target/classes/ 2>&1 | Out-
String
$jdepsOutput | Out-File -FilePath "jdeps-report.txt" -Encoding UTF8
Write-Host $jdepsOutput

# Module Dependencies
jdeps --print-module-deps target/classes/ 2>&1 | Out-File -FilePath "jdeps-
modules.txt" -Encoding UTF8

# JDK Internals (kritische Findings)
$internalsOutput = jdeps --jdk-internals target/classes/ 2>&1 | Out-String
$internalsOutput | Out-File -FilePath "jdeps-internals.txt" -Encoding UTF8

# Warnings parsen
$warnings = $jdepsOutput | Select-String "Warning:" | Measure-Object
Write-Host "Found $($warnings.Count) jdeps warnings" -ForegroundColor Yellow

# Kritische sun.* Package usage highlighten
if ($internalsOutput -match "sun\.") {
  Write-Host "CRITICAL: sun.* package usage detected!" -ForegroundColor Red
  $internalsOutput | Select-String "sun\." | ForEach-Object {
    Write-Host $_.Line -ForegroundColor Red
  }
}
```

### Was bedeutet das?

- **jdeps**: Java-Tool zur Analyse von Abhängigkeiten zwischen Klassen
- *sun. packages**: Interne Oracle-Klassen, die in JDK 9+ nicht mehr verfügbar sind
- **Warnings**: Potentielle Probleme bei JDK-Upgrade

### 3.2 Entfernte APIs erkennen (Removed APIs Detection):

### 🐧 Linux/macOS:

```
# APIs die in JDK 11+ entfernt wurden
echo "=== Checking for removed APIs ==="

# JAXB (removed in JDK 11)
grep -r "javax\.xml\.bind" --include="*.java" src/

# Java EE APIs (removed in JDK 11)
grep -r "javax\.annotation" --include="*.java" src/
grep -r "javax\.xml\.ws" --include="*.java" src/
```

### 🪟 Windows PowerShell:

```
# APIs die in JDK 11+ entfernt wurden
Write-Host "=== Checking for removed APIs ===" -ForegroundColor Cyan

# JAXB (removed in JDK 11)
$jaxbUsage = Get-ChildItem -Recurse -Include "*.java" -Path "src/" |
  Select-String "javax\.xml\.bind" | Measure-Object
Write-Host "JAXB usage found: $($jaxbUsage.Count) occurrences" -ForegroundColor
Yellow

# Java EE APIs (removed in JDK 11)
$annotationUsage = Get-ChildItem -Recurse -Include "*.java" -Path "src/" |
  Select-String "javax\.annotation" | Measure-Object
Write-Host "javax.annotation usage: $($annotationUsage.Count) occurrences" -
ForegroundColor Yellow

# Deprecated since JDK 8 (warning signs)
$dateUsage = Get-ChildItem -Recurse -Include "*.java" -Path "src/" |
  Select-String "new Date\(\)" | Measure-Object
Write-Host "Deprecated Date() usage: $($dateUsage.Count) occurrences" -
ForegroundColor Yellow

$calendarUsage = Get-ChildItem -Recurse -Include "*.java" -Path "src/" |
  Select-String "Calendar\.getInstance" | Measure-Object
Write-Host "Calendar usage: $($calendarUsage.Count) occurrences" -
ForegroundColor Yellow

# Summary Report
if ($jaxbUsage.Count -gt 0 -or $annotationUsage.Count -gt 0) {
  Write-Host "`nCRITICAL: Removed APIs detected - dependencies needed for JDK
11+" -ForegroundColor Red
}
```

### Was bedeutet das?

- **JAXB (Java Architecture for XML Binding)**: XML-zu-Java-Mapping, entfernt in JDK 11
- **javax.annotation**: Annotations wie @PostConstruct, entfernt in JDK 11
- **Date()**: Alte Date-API, deprecated seit JDK 8, sollte durch LocalDate ersetzt werden

## 📋 Bewertungs-Scorecard (Assessment Scorecard) - DETAILLIERTE BEWERTUNG

**Bewerte dein System basierend auf den Bewertungsergebnissen (Assessment Results):**

**Code-Qualität (Gewichtung: 40%):**

□ **Codezeilen (Lines of Code)** (___)
**5 = <50k LOC** - Klein, überschaubar, 6-8 Wochen Migration
**4 = 50-100k LOC** - Mittel, gut manageable, 8-12 Wochen
**3 = 100-200k LOC** - Groß, braucht Planung, 12-16 Wochen
**2 = 200-500k LOC** - Enterprise-Level, 16-24 Wochen
**1 = >500k LOC** - Monster-System, 24+ Wochen

□ **Test Coverage** (___)
**5 = >85%** - Excellente Coverage, geringes Regression-Risk
**4 = 70-85%** - Gute Coverage, akzeptables Risk
**3 = 50-70%** - Mittlere Coverage, mehr Tests nötig
**2 = 30-50%** - Schwache Coverage, Characterization Tests critical
**1 = <30%** - Katastrophale Coverage, Migration sehr risky

□ **Zyklomatische Komplexität (Cyclomatic Complexity)** (___)
**5 = All methods <10** - Clean Code, easy refactoring
**4 = Most methods <10, few >15** - Good structure, minor refactoring
**3 = Some methods 10-20** - Moderate complexity, refactoring needed
**2 = Many methods >20** - High complexity, significant refactoring
**1 = Multiple methods >30** - Monster methods, major refactoring required

**Abhängigkeits-Gesundheit (Dependency Health) (Gewichtung: 30%):**

□ **Veraltete Abhängigkeiten (Outdated Dependencies)** (___)
**5 = <10% outdated** - Modern dependency stack
**4 = 10-20% outdated** - Good maintenance, minor updates
**3 = 20-30% outdated** - Some technical debt, planned updates
**2 = 30-50% outdated** - Significant technical debt
**1 = >50% outdated** - Legacy dependency hell

□ **Sicherheitslücken (Security Vulnerabilities)** (___)
**5 = 0 CVEs** - Secure, production-ready
**4 = 1 Low CVE** - Minor security issue
**3 = 2-5 Medium CVEs** - Moderate security risk
**2 = 5-10 High CVEs** - Significant security risk
**1 = >10 Critical CVEs** - Security nightmare, immediate action needed

□ **JDK-Inkompatibilitäten (JDK Incompatibilities)** (___)
**5 = Clean jdeps report** - JDK 17 ready
**4 = 1-2 minor warnings** - Easy fixes
**3 = Few deprecated API warnings** - Some work needed
**2 = Multiple compatibility issues** - Significant effort required
**1 = Major compatibility blockers** - Extensive rework needed

**Team-Bereitschaft (Team Readiness) (Gewichtung: 30%):**

□ **JDK-17-Knowledge** (___)
**5 = Team knows modern Java** - Records, Pattern Matching, etc.
**4 = Good Java knowledge** - Familiar with JDK 11+ features
**3 = Some modern Java** - Basic understanding of newer features
**2 = Mostly JDK 8 knowledge** - Need training on modern features
**1 = Only legacy Java** - Extensive training required

□ **Testing-Culture** (___)
**5 = TDD, high automation** - Test-first mentality
**4 = Good unit testing** - Most code covered by tests
**3 = Some unit tests** - Basic testing practices
**2 = Manual testing focus** - Limited automated tests
**1 = No testing culture** - Manual testing only

□ **DevOps-Maturity** (___)
**5 = Full CI/CD automation** - Automated builds, tests, deployments
**4 = Advanced CI/CD** - Good automation, some manual steps
**3 = Basic CI/CD** - Automated builds and tests
**2 = Basic CI only** - Manual deployments
**1 = Manual everything** - No automation

**Gesamt-Score:** ___/45 Punkte

## 🎯 Score-Interpretation & Aktionsplan

**38-45 Punkte: 🟢 GRÜNES LICHT - Premium Migration**

- **Timeline:** 6-10 Wochen
- **Risk Level:** LOW
- **Strategy:** Aggressive timeline, modern features focus
- **Team Allocation:** 1-2 Senior Developers
- **Success Probability:** 95%

**Aktionspunkte (Action Items):**

- ✅ Start immediately with Phase 1
- ✅ Plan for modern Java features (Records, Pattern Matching)
- ✅ Consider JDK 8 → 17 direct jump
- ✅ Schedule post-migration optimization phase

**30-37 Punkte: 🟡 GELBES LICHT - Standard Migration**

- **Timeline:** 10-14 Wochen
- **Risk Level:** MEDIUM
- **Strategy:** Methodical approach, step-by-step
- **Team Allocation:** 2-3 Developers
- **Success Probability:** 85%

**Aktionspunkte:**

- ⚠️ Focus on Characterization Tests first
- ⚠️ Plan JDK 8 → 11 → 17 stepped approach
- ⚠️ Schedule dependency updates before JDK migration
- ⚠️ Consider team training on modern Java

**22-29 Punkte: 🟠 ORANGES LICHT - Hochrisiko-Migration (High-Risk Migration)**

- **Timeline:** 14-20 Wochen
- **Risk Level:** HIGH
- **Strategy:** Extensive preparation, conservative approach
- **Team Allocation:** 3-4 Developers + external support
- **Success Probability:** 70%

**Aktionspunkte:**

- 🚨 Major refactoring required before migration
- 🚨 Extensive Characterization Testing phase
- 🚨 Security vulnerabilities must be fixed first
- 🚨 Team training on testing and modern practices essential

**<22 Punkte: 🔴 ROTES LICHT - Migration nicht empfohlen**

- **Timeline:** 20+ Wochen (or project failure)
- **Risk Level:** VERY HIGH
- **Strategy:** Fix fundamentals first, then reconsider
- **Team Allocation:** Full team + external experts
- **Success Probability:** <50%

**Aktionspunkte:**

- 🛑 **DO NOT START JDK migration yet**
- 🛑 Focus on code quality improvements first
- 🛑 Establish testing culture and practices
- 🛑 Resolve security vulnerabilities
- 🛑 Consider bringing in external JDK migration experts
- 🛑 Plan 3-6 month preparation phase before migration

# 🧪 Phase 1: Characterization Tests - Das Sicherheitsnetz

## 🎯 Warum Characterization Tests?

**Problem:** Legacy-Code ohne Tests ist **unmigrierbar**.
**Lösung:** Tests schreiben, die **aktuelles Verhalten dokumentieren** (auch wenn es merkwürdig ist).

**Characterization Tests dokumentieren:**

- ✅ **Was der Code tut** (nicht was er tun sollte)
- ✅ **Grenzfälle (Edge Cases)** und merkwürdige Geschäftsregeln (Business Rules)
- ✅ **Performance-Charakteristiken**
- ✅ **Fehlerbehandlungs-Verhalten (Error Handling Behavior)**

## 📋 Characterization-Test-Strategie

### Schritt 1: Identifikation kritischer Pfade (Critical Path Identification)

```
// Finde die wichtigsten Code-Pfade:
// 1. Payment Processing (Zahlungsverarbeitung)
// 2. User Authentication (Benutzerauthentifizierung)
// 3. Data Import/Export (Datenimport/-export)
// 4. Reporting/Analytics (Berichte/Analysen)
// 5. Integration Points (APIs, Databases) (Integrationspunkte)
```

### Schritt 2: Input/Output Dokumentation

```
@Test
@DisplayName("PaymentProcessor should behave exactly like the legacy system")
class PaymentProcessorCharacterizationTest {

    // Dokumentiere AKTUELLES Verhalten, auch wenn es falsch erscheint
    @Test
    void shouldProcessWeekendGoldVipPayment() {
        // Given: Bekannte Eingabe aus Produktionsprotokollen (Production Logs)
        PaymentRequest request = PaymentRequest.builder()
            .paymentMethod("CREDIT_CARD")
            .amount(new BigDecimal("1500.00"))
            .userId(123L)
            .timestamp(getWeekendTimestamp()) // Samstag
            .build();

        User user = createGoldVipUser(123L);

        // When: Verarbeitung genau wie in der Produktion
        PaymentResult result = paymentProcessor.processPayment(request);

        // Then: Dokumentiere EXAKTES aktuelles Verhalten
        assertThat(result.isSuccess()).isTrue();
        assertThat(result.getProcessingFee()).isEqualTo(new
BigDecimal("15.50"));
        assertThat(result.getApprovalCode()).startsWith("WKND_GOLD_");
        assertThat(result.getProcessingTimeMs()).isBetween(50L, 200L);

        // Dokumentiere die merkwürdige Geschäftsregel
        assertThat(result.getNote()).contains("Weekend surcharge applied");
    }

    @Test
```

```
    void shouldFailSilverVipWeekendHighAmount() {
        // Dies dokumentiert eine MERKWÜRDIGE Geschäftsregel:
        // Silver VIP schlägt an Wochenenden für Beträge >1000€ fehl
        // Scheint wie ein Bug, aber es ist aktuelles Verhalten!

        PaymentRequest request = PaymentRequest.builder()
            .paymentMethod("CREDIT_CARD")
            .amount(new BigDecimal("1200.00"))
            .userId(456L)
            .timestamp(getWeekendTimestamp())
            .build();

        User user = createSilverVipUser(456L);

        PaymentResult result = paymentProcessor.processPayment(request);

        // Dokumentiere aktuelles (merkwürdiges) Verhalten
        assertThat(result.isSuccess()).isFalse();

assertThat(result.getErrorCode()).isEqualTo("SILVER_WEEKEND_RESTRICTION");
        assertThat(result.getErrorMessage()).contains("Silver VIP weekend limit
exceeded");
    }
}
```

### Schritt 3: Grenzfälle aus Produktionsprotokollen (Edge Cases from Production Logs)

```
@Test
void shouldHandleRealProductionEdgeCases() {
    // Sammle echte Eingaben aus Produktionsprotokollen:
    // - Null-Werte an unerwarteten Stellen
    // - Sonderzeichen in Namen
    // - Extreme Beträge (0.01€, 999999€)
    // - Ungültige aber akzeptierte Kreditkartenformate
    // - Zeitzone-Grenzfälle

    List<PaymentRequest> productionEdgeCases = loadFromProductionLogs();

    for (PaymentRequest request : productionEdgeCases) {
        // Dokumentiere: Keine Ausnahmen (Exceptions) sollten geworfen werden
        assertDoesNotThrow(() -> {
            PaymentResult result = paymentProcessor.processPayment(request);
            // Was auch immer das Ergebnis ist, dokumentiere es
            System.out.println("Input: " + request + " -> Output: " + result);
        });
    }
}
```

## 🛠️ Characterization-Test-Werkzeuge (Tools)

### Approval Testing (Golden Master):

```
@Test
void shouldProduceIdenticalOutputToGoldenMaster() {
    // Für komplexe Ausgaben (Reports, JSONs, etc.)
    String input = loadTestInput("payment-batch-1000-records.json");

    String actualOutput = paymentBatchProcessor.process(input);

    // Erster Lauf: Erstellt Golden Master Datei
    // Folgende Läufe: Vergleicht gegen Golden Master
    Approvals.verify(actualOutput);
}
```

**Performance Charakterisierung:**

```
@Test
void shouldMaintainCurrentPerformanceCharacteristics() {
    List<PaymentRequest> batchOf1000 = createLargeBatch(1000);

    long startTime = System.currentTimeMillis();
    paymentProcessor.processBatch(batchOf1000);
    long endTime = System.currentTimeMillis();

    long processingTimeMs = endTime - startTime;

    // Dokumentiere aktuelle Performance-Baseline
    assertThat(processingTimeMs).isLessThan(5000); // Aktuell dauert es ~3-4
Sekunden
    System.out.println("Batch processing baseline: " + processingTimeMs + "ms");
}
```

### 📊 Characterization-Test-Coverage-Ziel

**Minimum Coverage pro Komponente:**

- **Kritische Geschäftslogik (Critical Business Logic):** 100% (Payment, Authentication)
- **Integrationspunkte (Integration Points):** 90% (APIs, Database)
- **Hilfsprogramm-Klassen (Utility Classes):** 70%
- **Konfiguration/Setup:** 50%

**Werkzeuge für Coverage-Messung:**

```
# JaCoCo Coverage Report
mvn clean test jacoco:report
# Öffne: target/site/jacoco/index.html
# Ziel: >80% Gesamtcoverage für kritische Module
```

## 🔧 Phase 2: Dependency-Detox (Abhängigkeits-Entgiftung)

### 🎯 Warum Abhängigkeits-Updates VOR JDK-Migration?

**Grund:** Viele Dependencies haben **JDK-spezifische Versionen**. Alte Dependencies können JDK 17 **komplett blockieren**.

**Strategie: Sicherheitskritisch (Security-Critical) zuerst**, dann Kompatibilitätskritisch (Compatibility-Critical), dann Nice-to-Have.

### 📋 Dependency-Update-Fahrplan (Roadmap)

**Schritt 1: Sicherheitslücken-Triage (Security Vulnerability Triage)**

```
# OWASP Dependency Check
mvn org.owasp:dependency-check-maven:check
# Analysiere Report: target/dependency-check-report.html
# Priorität: CVE-Score > 7.0 = CRITICAL
```

**Kritische Sicherheitsupdates (sofort!):**

```
<!-- BEFORE: Security Nightmares -->
<log4j.version>1.2.17</log4j.version> <!-- 23+ CVEs! -->
<jackson.version>2.9.10</jackson.version> <!-- Multiple CVEs -->
```

```
<commons-lang.version>2.6</commons-lang.version> <!-- Ancient -->

<!-- AFTER: Secure Versions -->
<log4j.version>2.22.1</log4j.version> <!-- Latest, secure -->
<jackson.version>2.16.1</jackson.version> <!-- Current LTS -->
<commons-lang3.version>3.14.0</commons-lang3.version> <!-- Note: Lang3! -->
```

### Schritt 2: JDK-Kompatibilitäts-Updates

```
<!-- Dependencies with JDK 17 compatibility -->
<spring.version>5.3.30</spring.version> <!-- JDK 17 compatible -->
<hibernate.version>5.6.15.Final</hibernate.version> <!-- JDK 17 ready -->
<junit.version>5.10.1</junit.version> <!-- JUnit 5 for modern testing -->
```

### Schritt 3: Breaking-Change-Management

### commons-lang 2.x → 3.x Migration:

```
// Breaking Change: Package rename
// BEFORE:
import org.apache.commons.lang.StringUtils;
import org.apache.commons.lang.StringEscapeUtils;

// AFTER:
import org.apache.commons.lang3.StringUtils;
import org.apache.commons.text.StringEscapeUtils; // Moved to commons-text!
```

### Was bedeutet das?

- **Package Rename**: Der Paketname hat sich geändert - alle Imports müssen aktualisiert werden
- **commons-text**: Einige Funktionen wurden in ein separates Paket ausgelagert
- **Breaking Change**: Änderung, die bestehenden Code kaputt macht

### Log4j 1.x → 2.x Migration:

```
# BEFORE: log4j.properties
log4j.rootLogger=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

<!-- AFTER: log4j2.xml -->
<Configuration>
  <Appenders>
    <Console name="stdout" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg
%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="stdout"/>
    </Root>
  </Loggers>
</Configuration>
```

### Was bedeutet das?

- **Konfigurationsformat geändert**: Von Properties-Datei zu XML
- **API-Änderungen**: Andere Klassennamen und Methoden
- **Migration nötig**: Alle Log4j-Konfigurationen müssen angepasst werden

## 🧪 Dependency-Update-Test-Strategie

**Ein-nach-dem-anderen-Updates (One-by-One Updates):**

```
# NICHT alle auf einmal updaten!
# Strategie: Ein Dependency nach dem anderen

# 1. Log4j updaten
git checkout -b feature/update-log4j
# Update nur log4j, testen, committen

# 2. Commons-lang updaten
git checkout -b feature/update-commons-lang
# Update nur commons-lang, testen, committen

# 3. Jackson updaten
git checkout -b feature/update-jackson
# etc.
```

**Warum einzeln?**

- **Isolierung von Problemen**: Wenn etwas schief geht, weißt du genau welche Abhängigkeit schuld ist
- **Einfacheres Rollback**: Du kannst einzelne Updates rückgängig machen
- **Bessere Code-Review**: Kleinere Änderungen sind einfacher zu überprüfen

**Regression-Testing nach jedem Update:**

```
# Full Test Suite nach jedem Dependency-Update
mvn clean test

# Integration Tests
mvn clean verify -P integration-tests

# Performance Regression Check
mvn clean test -P performance-tests
```

## ⚠️ Dependency-Hell-Troubleshooting

**Problem: Transitive Dependency Conflicts (Konflikte transitiver Abhängigkeiten)**

```
<!-- Solution: Explicit Exclusions -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>5.3.30</version>
  <exclusions>
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

**Was bedeutet das?**

- **Transitive Dependencies**: Abhängigkeiten von deinen Abhängigkeiten
- **Exclusions**: Du schließt bestimmte transitive Abhängigkeiten explizit aus
- **Konfliktlösung**: Verhindert Versionskonflikte zwischen verschiedenen Bibliotheken

**Problem: Inkompatible API-Änderungen**

```
// Strategie: Adapter Pattern für Breaking Changes
public class CommonsLangAdapter {

    // Wrapper für alte API Calls
    public static boolean isNotEmpty(String str) {
        // Intern wird commons-lang3 verwendet, aber alte API bereitgestellt
        return org.apache.commons.lang3.StringUtils.isNotEmpty(str);
    }
}

// Usage in Legacy Code:
// BEFORE: org.apache.commons.lang.StringUtils.isNotEmpty(str)
// AFTER: CommonsLangAdapter.isNotEmpty(str)
```

**Was bedeutet das?**

- **Adapter Pattern**: Designmuster, das inkompatible APIs kompatibel macht
- **Wrapper**: Eine Hülle um die neue API, die die alte API nachahmt
- **Schrittweise Migration**: Du kannst die alte API beibehalten und intern die neue verwenden

# ⚔️ Phase 3: Code-Refactoring - Monster-Zähmung

## 🎯 Warum Refactoring VOR JDK-Migration?

**Monster-Methoden** (Complexity >20) sind schwer zu migrieren:

- **Debugging** ist ein Albtraum bei Breaking Changes
- **Rollback** ist unmöglich bei 200-Zeilen-Methoden
- **Risiko** ist exponentiell höher

**Strategie: Extract Method** bis alle Methoden <20 Zeilen sind.

## 📋 Monster-Method-Identifikation

**SonarQube-basierte Priorisierung:**

```
# SonarQube Complexity Report
mvn sonar:sonar
# In SonarQube UI:
# Issues → Type: Code Smell → Rule: "Cyclomatic Complexity"
# Sortiere nach Complexity DESC
```

**Command-Line-Alternative:**

```
# Komplexe Methoden mit grep finden
find . -name "*.java" -exec grep -l "if\|for\|while\|switch" {} \; | \
xargs -I {} sh -c 'echo "=== {} ==="; grep -c "if\|for\|while\|switch" "{}"' | \
sort -rn
```

**Was bedeutet das?**

- **Cyclomatic Complexity**: Maß für die Komplexität von Code (Anzahl unabhängiger Pfade)
- **Code Smell**: Code der funktioniert, aber schlecht strukturiert ist
- **Komplexe Methoden**: Methoden mit vielen if/for/while-Statements

## ⚔️ Monster-Method-Refactoring-Strategie

**Beispiel: PaymentProcessor-Bestie (Complexity 47)**

### VORHER: 147-Zeilen-Monster

```
public PaymentResult processPayment(PaymentRequest request) {
    if (request != null) {
        if (request.getAmount() != null) {
            if (request.getAmount().compareTo(BigDecimal.ZERO) > 0) {
                if (request.getPaymentMethod() != null) {
                    if ("CREDIT_CARD".equals(request.getPaymentMethod())) {
                        if (request.getCreditCard() != null) {
                            // ... 140 more lines of horror
                            for (int i = 0; i < retryCount; i++) {
                                try {
                                    if (isWeekend()) {
                                        if (amount.compareTo(new
BigDecimal("1000")) > 0) {
                                            if (user.getVipStatus() ==
VipStatus.GOLD) {
                                                // Weekend + High Amount + Gold
VIP logic
                                            } else if (user.getVipStatus() ==
VipStatus.SILVER) {
                                                // Weekend + High Amount +
Silver VIP logic
                                            }
                                        }
                                    }
                                } catch (PaymentException e) {
                                    // Retry logic
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return new PaymentResult(false, "Invalid request");
}
```

### NACHHER: Saubere Architektur (Clean Architecture)

```
public PaymentResult processPayment(PaymentRequest request) {
    // Schritt 1: Validierung
    PaymentValidationResult validation = validatePaymentRequest(request);
    if (!validation.isValid()) {
        return PaymentResult.failure(validation.getErrorMessage());
    }

    // Schritt 2: Geschäftsregeln
    PaymentRuleResult ruleResult = paymentRulesEngine.applyRules(request);
    if (!ruleResult.isAllowed()) {
        return PaymentResult.failure(ruleResult.getReasonForDenial());
    }

    // Schritt 3: Verarbeitung
    PaymentMethodProcessor processor =
getProcessorFor(request.getPaymentMethod());
    return processor.process(request);
}
```

```
// Extrahierte Methoden (jede <20 Zeilen):
private PaymentValidationResult validatePaymentRequest(PaymentRequest request) {
/* ... */ }
private PaymentMethodProcessor getProcessorFor(String paymentMethod) { /* ... */
}
```

**Was wurde verbessert?**

- **Lesbarkeit**: Code ist jetzt auf einen Blick verständlich
- **Testbarkeit**: Jede Methode kann einzeln getestet werden
- **Wartbarkeit**: Änderungen sind isoliert und sicher
- **Debuggbarkeit**: Probleme können schneller lokalisiert werden

## 🧩 Extract Method Techniken

### Technik 1: Validierungs-Logik extrahieren (Extract Validation Logic)

```
// VORHER: Validierung in Geschäftslogik versteckt
if (request != null && request.getAmount() != null &&
    request.getAmount().compareTo(BigDecimal.ZERO) > 0 &&
    request.getPaymentMethod() != null && /* 10 weitere Bedingungen */) {
    // business logic
}

// NACHHER: Saubere Validierung
private PaymentValidationResult validatePaymentRequest(PaymentRequest request) {
    if (request == null) {
        return PaymentValidationResult.invalid("Request cannot be null");
    }

    if (request.getAmount() == null ||
request.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
        return PaymentValidationResult.invalid("Amount must be positive");
    }

    if (StringUtils.isBlank(request.getPaymentMethod())) {
        return PaymentValidationResult.invalid("Payment method is required");
    }

    return PaymentValidationResult.valid();
}
```

### Technik 2: Geschäftsregeln extrahieren (Extract Business Rules)

```
// VORHER: Geschäftsregeln in if/else-Hölle verstreut
if (isWeekend()) {
    if (amount.compareTo(new BigDecimal("1000")) > 0) {
        if (user.getVipStatus() == VipStatus.GOLD) {
            fee = amount.multiply(new BigDecimal("0.015"));
        } else {
            return PaymentResult.failure("Silver VIP weekend restriction");
        }
    }
}

// NACHHER: Explizite Geschäftsregeln
private PaymentRuleResult applyWeekendRules(PaymentRequest request, User user) {
    if (!isWeekend()) {
        return PaymentRuleResult.allowed();
    }
```

```
    if (request.getAmount().compareTo(new BigDecimal("1000")) <= 0) {
        return PaymentRuleResult.allowed();
    }

    if (user.getVipStatus() == VipStatus.GOLD) {
        return PaymentRuleResult.allowedWithSurcharge(new BigDecimal("0.015"));
    }

    return PaymentRuleResult.denied("Silver VIP weekend restriction for amounts
>1000€");
}
```

### Technik 3: Strategy Pattern für Zahlungsmethoden (Payment Methods)

```
// VORHER: Massive if/else-Kette
if ("CREDIT_CARD".equals(paymentMethod)) {
    // 50 Zeilen Kreditkarten-Logik
} else if ("BANK_TRANSFER".equals(paymentMethod)) {
    // 40 Zeilen Bank-Transfer-Logik
} else if ("PAYPAL".equals(paymentMethod)) {
    // 30 Zeilen PayPal-Logik
}

// NACHHER: Strategy Pattern
public interface PaymentMethodProcessor {
    PaymentResult process(PaymentRequest request);
    boolean canHandle(String paymentMethod);
}

@Component
public class CreditCardProcessor implements PaymentMethodProcessor {
    @Override
    public PaymentResult process(PaymentRequest request) {
        // Saubere, fokussierte Kreditkarten-Logik (20 Zeilen max)
    }

    @Override
    public boolean canHandle(String paymentMethod) {
        return "CREDIT_CARD".equals(paymentMethod);
    }
}

// Zahlungsmethoden-Routing
private PaymentMethodProcessor getProcessorFor(String paymentMethod) {
    return processors.stream()
        .filter(p -> p.canHandle(paymentMethod))
        .findFirst()
        .orElseThrow(() -> new
UnsupportedPaymentMethodException(paymentMethod));
}
```

### Was ist das Strategy Pattern?

- **Designmuster**: Verschiedene Algorithmen (Strategien) können zur Laufzeit ausgewählt werden
- **Erweiterbarkeit**: Neue Zahlungsmethoden können ohne Änderung bestehenden Codes hinzugefügt werden
- **Trennung der Verantwortlichkeiten**: Jede Zahlungsmethode hat ihre eigene Klasse

### 🧪 Refactoring-Test-Strategie

**Goldene Regel:** Nach jedem Extract Method → Tests laufen lassen!

```
# Nach jeder Refactoring-Iteration
mvn clean test
# Characterization Tests müssen IMMER grün bleiben
# Falls nicht: Rollback und kleinere Schritte machen
```

**Test Coverage während Refactoring:**

```
@Test
void shouldMaintainSameBehaviorAfterRefactoring() {
    // Golden Master Test: Output muss identisch bleiben
    PaymentRequest request = createComplexPaymentRequest();

    PaymentResult result = paymentProcessor.processPayment(request);

    // Diese Assertion darf sich NIEMALS ändern während Refactoring
    assertThat(result.isSuccess()).isTrue();
    assertThat(result.getProcessingFee()).isEqualTo(new BigDecimal("15.50"));
    assertThat(result.getApprovalCode()).startsWith("WKND_GOLD_");
}
```

## ☕ Phase 4: JDK-Migration - Der eigentliche Spaß!

### 🎯 Migrations-Strategie: JDK 8 → 11 → 17

**Warum Zwischenschritt über JDK 11?**

- ✅ **JDK 11 ist LTS** (Long Term Support bis 2032)
- ✅ **Weniger Breaking Changes** als direkter 8→17 Sprung
- ✅ **Bessere Kompatibilität** mit Legacy Dependencies
- ✅ **Rollback-Möglichkeit** falls JDK 17 Probleme macht

### 📋 JDK 8 → 11 Migration

**Schritt 1: Build Configuration Update**

```
<!-- pom.xml -->
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
  <java.version>11</java.version>
</properties>

<!-- Compiler Plugin Update -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.11.0</version>
  <configuration>
    <source>11</source>
    <target>11</target>
    <release>11</release>
  </configuration>
</plugin>
```

**Schritt 2: Deprecated API Fixes**

```
// VORHER: JDK 8 deprecated APIs
Date date = new Date(); // Deprecated since JDK 1.1!
Calendar cal = Calendar.getInstance();

// NACHHER: Modern Time API (verfügbar seit JDK 8!)
LocalDate date = LocalDate.now();
LocalDateTime dateTime = LocalDateTime.now();
```

**Was bedeutet das?**

- **Deprecated**: Als veraltet markiert, wird in zukünftigen Versionen entfernt
- **Modern Time API**: Neue, bessere API für Datum und Zeit seit JDK 8
- **LocalDate/LocalDateTime**: Unveränderliche (immutable) Klassen, thread-safe

**Schritt 3: JDK 11 Testing & Validation**

```
# Vollständige Test Suite
mvn clean test

# Performance Baseline neu messen
java -XX:+PrintGC -XX:+PrintGCDetails -jar target/app.jar

# Memory Usage prüfen
jstat -gc <PID> 1s 10

# Startup Time messen
time java -jar target/app.jar --spring.main.lazy-initialization=true
```

**Was messen diese Befehle?**

- **PrintGC**: Zeigt Garbage Collection Aktivität
- **jstat -gc**: Speicher-Statistiken der Garbage Collection
- **time**: Misst wie lange das Programm zum Starten braucht

## 📋 JDK 11 → 17 Migration

**Schritt 1: Build Configuration Update**

```
<properties>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
  <java.version>17</java.version>
</properties>
```

**Schritt 2: Moderne Java Features nutzen**

**Records für DTOs:**

```
// VORHER: Traditional DTO (50+ Zeilen)
public class PaymentRequest {
    private final String paymentMethod;
    private final BigDecimal amount;
    private final Long userId;
    private final LocalDateTime timestamp;

    public PaymentRequest(String paymentMethod, BigDecimal amount, Long userId,
LocalDateTime timestamp) {
        this.paymentMethod = paymentMethod;
        this.amount = amount;
        this.userId = userId;
        this.timestamp = timestamp;
```

```
    }

    // 40+ Zeilen getters, equals, hashCode, toString
}

// NACHHER: Record (1 Zeile!)
public record PaymentRequest(
    String paymentMethod,
    BigDecimal amount,
    Long userId,
    LocalDateTime timestamp
) {}
```

**Was sind Records?**

- **Kompakte Datenklassen**: Automatische Generierung von getters, equals, hashCode, toString
- **Unveränderlich (Immutable)**: Werte können nach Erstellung nicht geändert werden
- **Weniger Boilerplate**: Viel weniger Code für einfache Datenstrukturen

**Text Blocks für komplexe Strings:**

```
// VORHER: String concatenation horror
String emailTemplate = "Dear " + user.getName() + ",\n\n" +
    "Your payment of " + amount + " has been processed.\n" +
    "Transaction ID: " + transactionId + "\n\n" +
    "Best regards,\n" +
    "Payment Team";

// NACHHER: Text block elegance
String emailTemplate = """
    Dear %s,

    Your payment of %s has been processed.
    Transaction ID: %s

    Best regards,
    Payment Team
    """.formatted(user.getName(), amount, transactionId);
```

**Was sind Text Blocks?**

- **Mehrzeilige Strings**: Einfache Definition von mehrzeiligen Texten
- **Keine Escape-Zeichen**: Anführungszeichen und Zeilenumbrüche funktionieren natürlich
- **Bessere Lesbarkeit**: Code ist viel sauberer und verständlicher

**Pattern Matching für instanceof:**

```
// VORHER: instanceof casting dance
if (paymentMethod instanceof CreditCardMethod) {
    CreditCardMethod ccMethod = (CreditCardMethod) paymentMethod;
    return processCreditCard(ccMethod.getCardNumber(), ccMethod.getCvv());
} else if (paymentMethod instanceof BankTransferMethod) {
    BankTransferMethod btMethod = (BankTransferMethod) paymentMethod;
    return processBankTransfer(btMethod.getIban(), btMethod.isExpress());
}

// NACHHER: Pattern matching elegance
if (paymentMethod instanceof CreditCardMethod(var cardNumber, var cvv)) {
    return processCreditCard(cardNumber, cvv);
```

```
} else if (paymentMethod instanceof BankTransferMethod(var iban, var isExpress))
{
    return processBankTransfer(iban, isExpress);
}
```

**Was ist Pattern Matching?**

- **Typ-Check und Casting in einem**: Keine separaten Casting-Schritte mehr nötig
- **Automatische Variablen-Extraktion**: Werte werden direkt extrahiert
- **Weniger fehleranfällig**: Kein manuelles Casting mehr

**Switch Expressions:**

```
// VORHER: Traditional switch statement
String feeCategory;
switch (user.getVipStatus()) {
    case GOLD:
        feeCategory = "premium";
        break;
    case SILVER:
        feeCategory = "standard";
        break;
    case BRONZE:
        feeCategory = "basic";
        break;
    default:
        feeCategory = "regular";
        break;
}

// NACHHER: Switch expression
String feeCategory = switch (user.getVipStatus()) {
    case GOLD -> "premium";
    case SILVER -> "standard";
    case BRONZE -> "basic";
    default -> "regular";
};
```

**Was sind Switch Expressions?**

- **Werte zurückgeben**: Switch kann direkt einen Wert zurückgeben
- **Kein break nötig**: Mit `->` syntax kein break statement erforderlich
- **Kompakter**: Weniger Code, bessere Lesbarkeit
- **Exhaustive**: Compiler prüft, dass alle Fälle abgedeckt sind

## ⚠️ JDK 17 Migration Troubleshooting

**Problem: Reflection Access Restrictions (Reflexions-Zugriffs-Beschränkungen)**

```
# Error: Unable to make field accessible
# Solution: Add JVM arguments
--add-opens java.base/java.lang=ALL-UNNAMED
--add-opens java.base/java.util=ALL-UNNAMED
--add-exports java.base/sun.security.util=ALL-UNNAMED
```

**Was bedeutet das?**

- **Reflection**: Java-Feature zum dynamischen Zugriff auf Klassen zur Laufzeit
- **--add-opens**: Öffnet interne Module für Reflection

- **ALL-UNNAMED**: Erlaubt Zugriff von unbenannten Modulen (dein Code)
- **Temporäre Lösung**: Besser ist es, Frameworks zu aktualisieren, die diese Beschränkungen respektieren

**Problem: Entfernte Tools und APIs (Removed Tools and APIs)**

```
// ENTFERNT in JDK 17: Nashorn JavaScript Engine
// VORHER:
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// NACHHER: Use GraalVM JavaScript or external library
// Add dependency: org.graalvm.js:js:22.3.0
```

**Was bedeutet das?**

- **Nashorn**: JavaScript-Engine die in JDK enthalten war
- **Entfernt in JDK 17**: Nicht mehr verfügbar
- **Alternative**: GraalVM JavaScript oder andere externe Bibliothek nötig

**Problem: SSL/TLS Änderungen**

```
// JDK 17 has stricter TLS requirements
// Solution: Update TLS configuration
System.setProperty("https.protocols", "TLSv1.2,TLSv1.3");
System.setProperty("jdk.tls.client.protocols", "TLSv1.2,TLSv1.3");
```

**Was bedeutet das?**

- **TLS (Transport Layer Security)**: Verschlüsselung für HTTPS-Verbindungen
- **Strengere Anforderungen**: JDK 17 erlaubt nur sichere TLS-Versionen
- **Alte Versionen deaktiviert**: TLSv1.0 und TLSv1.1 sind nicht mehr verfügbar

# 🚀 Phase 5: Modernisierung & Optimierung

## 🎯 Performance Optimierung

**JDK 17 GC Tuning (Garbage Collector Optimierung):**

```
# Modern GC Options (JDK 17)
# G1GC (default, good for most cases)
-XX:+UseG1GC
-XX:MaxGCPauseMillis=200
-XX:G1HeapRegionSize=16m

# ZGC (ultra-low latency)
-XX:+UseZGC
-XX:+UnlockExperimentalVMOptions

# Parallel GC (high throughput)
-XX:+UseParallelGC
-XX:ParallelGCThreads=8
```

**Was sind diese GCs?**

- **G1GC**: Standard Garbage Collector, gute Balance zwischen Latenz und Durchsatz
- **ZGC**: Ultra-niedrige Latenz, für latenz-kritische Anwendungen
- **Parallel GC**: Hoher Durchsatz, für batch-verarbeitende Anwendungen

- **MaxGCPauseMillis**: Maximale Pause-Zeit für Garbage Collection

**Memory Optimization (Speicher-Optimierung):**

```
# Modern Memory Settings
-Xms2g                    # Anfangsgröße des Heaps
-Xmx4g                    # Maximale Heap-Größe
-XX:+UseStringDeduplication  # Gleiche Strings teilen Speicher
-XX:+UseCompressedOops       # Kleinere Zeiger auf 64-Bit-Systemen
```

## 📊 Performance Monitoring Setup

**Application Performance Monitoring:**

```
// JDK 17+ JFR (Java Flight Recorder) Integration
@Component
public class PerformanceMonitor {

    @EventHandler
    public void onPaymentProcessed(PaymentProcessedEvent event) {
        // JFR Custom Events
        PaymentEvent jfrEvent = new PaymentEvent();
        jfrEvent.paymentMethod = event.getPaymentMethod();
        jfrEvent.processingTimeMs = event.getProcessingTime();
        jfrEvent.amount = event.getAmount();
        jfrEvent.commit();
    }
}

@Name("com.company.PaymentEvent")
@Label("Payment Processing")
public class PaymentEvent extends Event {
    @Label("Payment Method")
    public String paymentMethod;

    @Label("Processing Time")
    @Timespan(Timespan.MILLISECONDS)
    public long processingTimeMs;

    @Label("Amount")
    public BigDecimal amount;
}
```

**Was ist Java Flight Recorder (JFR)?**

- **Low-Overhead Profiling**: Sammelt Performance-Daten mit minimalem Performance-Impact
- **Production-Ready**: Kann sicher in Produktionsumgebungen verwendet werden
- **Custom Events**: Du kannst eigene Events für Business-Metriken definieren
- **Analyse-Tools**: JDK Mission Control kann JFR-Dateien analysieren

**Metrics Collection (Metriken-Sammlung):**

```
# Enable JFR for Production
-XX:+FlightRecorder
-XX:StartFlightRecording=duration=60s,filename=payment-app.jfr
-XX:FlightRecorderOptions=settings=profile
```

## 🧪 Moderne Tests mit JUnit 5

### Migration from JUnit 4 to 5:

```
// VORHER: JUnit 4
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;

public class PaymentProcessorTest {

    @Before
    public void setUp() {
        // setup
    }

    @Test
    public void shouldProcessPayment() {
        assertTrue(result.isSuccess());
    }
}

// NACHHER: JUnit 5
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import static org.junit.jupiter.api.Assertions.*;

class PaymentProcessorTest {

    @BeforeEach
    void setUp() {
        // setup
    }

    @Test
    @DisplayName("Should process valid credit card payment successfully")
    void shouldProcessPayment() {
        assertThat(result.isSuccess()).isTrue();

assertThat(result.getProcessingTime()).isLessThan(Duration.ofSeconds(1));
    }
}
```

### Was ist neu in JUnit 5?

- **@DisplayName**: Bessere, lesbare Testnamen
- **@BeforeEach/@AfterEach**: Klarere Namen statt @Before/@After
- **Bessere Assertions**: Mehr aussagekräftige Fehlermeldungen
- **Lambda-Support**: Tests können Lambdas verwenden

### Modern Testing Features:

```
// Parameterized Tests (Parametrisierte Tests)
@ParameterizedTest
@ValueSource(strings = {"CREDIT_CARD", "BANK_TRANSFER", "PAYPAL"})
@DisplayName("Should process payments for all supported payment methods")
void shouldProcessAllPaymentMethods(String paymentMethod) {
    PaymentRequest request = createRequestFor(paymentMethod);
    PaymentResult result = processor.processPayment(request);
    assertThat(result.isSuccess()).isTrue();
}
```

```
// Dynamic Tests (Dynamische Tests)
@TestFactory
Stream<DynamicTest> shouldHandleRealProductionData() {
    return loadProductionTestCases().stream()
        .map(testCase -> DynamicTest.dynamicTest(
            "Process " + testCase.getDescription(),
            () -> {
                PaymentResult result =
processor.processPayment(testCase.getRequest());
                assertEquals(testCase.getExpectedResult(), result);
            }
        ));
}
```

**Was sind diese Test-Features?**

- **Parameterized Tests**: Ein Test mit verschiedenen Eingabewerten
- **Dynamic Tests**: Tests die zur Laufzeit generiert werden
- **Test Factory**: Methode die mehrere Tests dynamisch erstellt

# 📊 Erfolgs-Metriken & Monitoring

## 🎯 Key Performance Indicators (Wichtige Leistungsindikatoren)

**Pre-Migration Baseline vs. Post-Migration:**

Performance Metrics to Track:

| Metric | JDK 8 | JDK 17 | Target |
|---|---|---|---|
| Startup Time | _____ sec | _____ sec | <60 sec |
| Memory Usage | _____ MB | _____ MB | -20% |
| GC Pause P99 | _____ ms | _____ ms | <100ms |
| Throughput | _____ req/s | _____ req/s | +15% |
| Build Time | _____ min | _____ min | -30% |
| Test Suite | _____ sec | _____ sec | -25% |

Code Quality Metrics:

| Quality Gate | Before | After | Target |
|---|---|---|---|
| Complexity | _____ avg | _____ avg | <10 |
| Test Coverage | _____ % | _____ % | >80% |
| Security Issues | _____ count | _____ count | 0 |
| Code Duplicatn | _____ % | _____ % | <3% |
| Tech Debt | _____ hours | _____ hours | -50% |

**Was bedeuten diese Metriken?**

- **Startup Time**: Zeit bis die Anwendung läuft
- **Memory Usage**: Speicherverbrauch
- **GC Pause P99**: 99% aller Garbage Collection Pausen sind kürzer
- **Throughput**: Requests pro Sekunde
- **Tech Debt**: Technische Schulden in Stunden geschätzt

## 🔍 Monitoring Setup

### Application Metrics:

```
@Component
public class JdkMigrationMetrics {

    private final MeterRegistry meterRegistry;
    private final Timer paymentProcessingTime;
    private final Counter paymentSuccessCount;
    private final Counter paymentFailureCount;

    public JdkMigrationMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.paymentProcessingTime = Timer.builder("payment.processing.time")
            .description("Payment processing duration")
            .register(meterRegistry);
        this.paymentSuccessCount = Counter.builder("payment.success.count")
            .description("Successful payments")
            .register(meterRegistry);
        this.paymentFailureCount = Counter.builder("payment.failure.count")
            .description("Failed payments")
            .register(meterRegistry);
    }

    public void recordPaymentProcessed(PaymentResult result, Duration
processingTime) {
        paymentProcessingTime.record(processingTime);

        if (result.isSuccess()) {
            paymentSuccessCount.increment();
        } else {
            paymentFailureCount.increment();
        }
    }
}
```

### Was ist das?

- **MeterRegistry**: Zentrale Stelle für alle Metriken
- **Timer**: Misst Ausführungszeiten
- **Counter**: Zählt Ereignisse
- **Micrometer**: Standard-Library für Metriken in Java

### JVM Metrics:

```
# Production JVM Monitoring
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false

# JFR for detailed profiling
-XX:+FlightRecorder
-XX:StartFlightRecording=duration=300s,filename=production-profile.jfr
```

# 🆘 Troubleshooting Guide (Fehlerbehebungshandbuch)

## 🚨 Häufige Probleme & Lösungen

### Problem 1: NoClassDefFoundError nach JDK-Upgrade

```
# Symptom:
java.lang.NoClassDefFoundError: javax/xml/bind/JAXBException

# Ursache: JAXB removed from JDK 11+
# Lösung: Add JAXB dependency

<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.1</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.8</version>
</dependency>
```

### Was bedeutet das?

- **NoClassDefFoundError**: Klasse wird zur Laufzeit nicht gefunden
- **JAXB**: XML-Binding-Framework, entfernt in JDK 11+
- **Lösung**: Explizit als Dependency hinzufügen

### Problem 2: Illegal Reflective Access

```
# Warning:
WARNING: Illegal reflective access by
org.springframework.cglib.core.ReflectUtils

# Lösung: Add JVM arguments (temporary)
--add-opens java.base/java.lang=ALL-UNNAMED
--add-opens java.base/java.lang.reflect=ALL-UNNAMED

# Langfristige Lösung: Framework-Updates

<spring.version>5.3.30</spring.version> <!-- JDK 17 compatible -->
```

### Problem 3: Performance Regression (Performance-Rückgang)

```
# Symptom: App läuft langsamer nach Migration
# Debug: JFR Profile vergleichen

# JDK 8 Profiling
java -XX:+FlightRecorder -XX:StartFlightRecording=duration=60s,filename=jdk8.jfr
-jar app.jar

# JDK 17 Profiling
java -XX:+FlightRecorder -
XX:StartFlightRecording=duration=60s,filename=jdk17.jfr -jar app.jar

# Analyse mit JDK Mission Control
jmc jdk8.jfr
jmc jdk17.jfr
```

**Was tun bei Performance-Problemen?**

- **JFR Vergleich**: Profile vor und nach Migration vergleichen
- **GC-Tuning**: Verschiedene Garbage Collectors ausprobieren
- **Memory Settings**: Heap-Größe anpassen
- **JDK Mission Control**: Profiling-Tool von Oracle

**Problem 4: Build-Failures**

```
<!-- Maven Compatibility Issues -->
<!-- Lösung: Update Maven & Plugins -->
<maven.version>3.9.4</maven.version>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.11.0</version>
</plugin>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.1.2</version>
</plugin>
```

**Problem 5: SSL/Certificate Issues**

```
// JDK 17 strengere Certificate-Validation
// Lösung: Certificate-Store updaten oder TLS-Config anpassen
System.setProperty("javax.net.ssl.trustStore", "/path/to/truststore.jks");
System.setProperty("javax.net.ssl.trustStorePassword", "changeit");
System.setProperty("https.protocols", "TLSv1.2,TLSv1.3");
```

## 🔧 Emergency Rollback Plan (Notfall-Rollback-Plan)

**Rollback-Strategy (falls alles schiefgeht):**

```
# 1. Git Rollback
git revert <migration-commit-hash>

# 2. Docker Image Rollback (falls containerized)
docker tag myapp:jdk8-backup myapp:latest
docker push myapp:latest

# 3. JVM Rollback
# Switch back to JDK 8 installation
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk

# 4. Dependency Rollback
# Revert pom.xml to JDK 8 compatible versions
git checkout HEAD~1 -- pom.xml

# 5. Database Schema Rollback (if needed)
# Revert any JDK 17 specific database changes
```

**Rollback Testing:**

```
# ALWAYS test rollback procedure BEFORE migration
# Create test environment with JDK 17 → rollback to JDK 8
# Ensure data compatibility and no data loss
```

**Warum Rollback-Plan wichtig ist:**

- **Murphy's Law**: Was schief gehen kann, wird schief gehen
- **Business Continuity**: Geschäft muss weiterlaufen
- **Stress-Reduktion**: Plan B reduziert Panik
- **Vertrauen**: Management vertraut mehr mit Rollback-Plan

# 🎉 Erfolgs-Feier & Nächste Schritte

## 🏆 Migrations-Abschluss-Checkliste

**Technische Verifikation:**

- ☐ **Alle Tests bestehen** auf JDK 17
- ☐ **Performance-Metriken** erreichen Ziele
- ☐ **Sicherheitslücken** behoben
- ☐ **Keine Regression-Probleme** in Produktion
- ☐ **Monitoring** zeigt gesunde Metriken

**Team & Prozess:**

- ☐ **Team-Training** zu JDK 17 Features abgeschlossen
- ☐ **Dokumentation** aktualisiert (README, Architektur-Docs)
- ☐ **Runbooks** für JDK 17 Betrieb aktualisiert
- ☐ **Deployment-Pipeline** funktioniert mit JDK 17

**Geschäftswert (Business Value):**

- ☐ **Performance-Verbesserungen** für Benutzer sichtbar
- ☐ **Security Compliance** Anforderungen erfüllt
- ☐ **Entwickler-Produktivität** gestiegen
- ☐ **Recruiting-Vorteil** (moderner Tech-Stack)

## 🚀 Post-Migrations-Möglichkeiten

**Moderne Java Features erkunden:**

- **Virtual Threads** (Project Loom - JDK 19+)
- **Pattern Matching** Verbesserungen (laufend)
- **Foreign Function & Memory API** (fortgeschrittene Anwendungsfälle)
- **Vector API** (performance-kritische Anwendungen)

**Architektur-Modernisierung:**

- **Microservices Migration** (mit modernem Java)
- **Cloud-Native Deployment** (GraalVM native images)
- **Reactive Programming** (Spring WebFlux mit modernem Java)

## 📚 Lernressourcen

**Bücher:**

- **"Modern Java Recipes"** - Ken Kousen
- **"Java: The Complete Reference, 12th Edition"** - Herbert Schildt
- **"Effective Java, 3rd Edition"** - Joshua Bloch

**Online Resources:**

- **Java Language Specification JDK 17:** https://docs.oracle.com/javase/specs/jls/se17/html/
- **Migration Guide:** https://docs.oracle.com/en/java/javase/17/migrate/
- **JDK 17 Features:** https://openjdk.org/projects/jdk/17/

**Community:**

- **Java Subreddit:** r/java
- **Stack Overflow:** java + jdk-17 tags
- **OpenJDK Mailing Lists:** https://mail.openjdk.org/

# 📞 Support & Feedback

**Fragen zu diesem Handbuch?**

📧 **Elyndra persönlich:** elyndra.valen@java-developer.online

**Besonders interessiert bin ich an:**

- **Euren Erfolgsgeschichten** (Performance-Gewinne, Team-Feedback)
- **Problemen die aufgetreten sind** (ich erweitere das Troubleshooting)
- **Verbesserungsvorschlägen** für das Handbuch
- **Spezifische Anwendungsfälle** (Enterprise-Einschränkungen, Legacy-Systeme)

## 🤝 Community-Beitrag

**Dieses Handbuch lebt von euren Erfahrungen!**

Schickt mir gerne:

- **Zeitschätzungen** aus euren Projekten
- **Weitere Troubleshooting-Fälle**
- **Tool-Empfehlungen** die geholfen haben
- **Lessons Learned** aus euren Migrationen

**Die besten Beiträge** fließen in die nächste Version des Handbuchs ein!

# 📋 Schnellreferenz-Karte (Quick Reference Card)

**Kopiere diese Seite für dein Team:**

## Migrations-Timeline (Richtwerte):

- **Small Project** (<50k LOC): 4-8 Wochen
- **Medium Project** (50-200k LOC): 8-12 Wochen
- **Large Project** (>200k LOC): 12-20 Wochen
- **Enterprise Monster** (>500k LOC): 20-30 Wochen

## Phasen-Aufschlüsselung:

- **Phase 0 (Assessment):** 5-10% der Zeit
- **Phase 1 (Tests):** 30-40% der Zeit
- **Phase 2 (Dependencies):** 10-15% der Zeit
- **Phase 3 (Refactoring):** 25-35% der Zeit
- **Phase 4 (Migration):** 5-10% der Zeit
- **Phase 5 (Modernization):** 10-15% der Zeit

## Essential Commands:

```
# Pre-Migration Assessment
jdeps --check-modules target/classes/
mvn org.owasp:dependency-check-maven:check
mvn sonar:sonar

# During Migration
mvn clean test
mvn clean verify -P integration-tests
java -XX:+FlightRecorder -
XX:StartFlightRecording=duration=60s,filename=profile.jfr -jar app.jar

# Post-Migration Validation
jstat -gc <PID> 1s 10
curl -s http://localhost:8080/actuator/health
```

## Emergency Contacts:

- **Build Issues:** Maven Central, Stack Overflow
- **JVM Issues:** OpenJDK Bug Database
- **Performance:** JDK Mission Control, VisualVM
- **Security:** OWASP, CVE Database

## 🚦 Go/No-Go Entscheidung

**35-45 Punkte:** 🟢 **Leg los!** (8-12 Wochen)
**25-34 Punkte:** 🟡 **Erst vorbereiten** (12-16 Wochen + Vorbereitung)
**15-24 Punkte:** 🟠 **Hohes Risiko** (16-24 Wochen, Team-Training erwägen)
**<15 Punkte:** 🔴 **Noch nicht starten** (Erst Grundlagen reparieren)

---

**Version 1.0 - August 2025**