Legacy-Code-Assessment-Kit

🏺 Systematische Bewertung und Modernisierungsstrategie

Von Elyndra Valen & Dr. Cassian Holt - Java Fleet Systems Consulting
Basiert auf 50+ Legacy-Modernisierungs-Projekten

📋 Quick Start - Assessment in 30 Minuten

Phase 1: Basis-Metriken sammeln (10 Min)

🐧 Linux/macOS (Bash):

```
# Lines of Code Analysis
find . -name "*.java" -not -path "*/target/*" -not -path "*/test/*" | xargs wc -
l | tail -1

# Komplexeste Dateien identifizieren (>500 LOC)
find . -name "*.java" -not -path "*/target/*" | xargs wc -l | sort -rn | head -
10

# Dependency Analysis
mvn dependency:tree | grep -E "compile|runtime" | wc -l
```

🪟 Windows (PowerShell):

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

# Komplexeste Dateien identifizieren (>500 LOC)
Get-ChildItem -Recurse -Include "*.java" -Exclude "*target*" |
    ForEach-Object {
        $lineCount = (Get-Content $_.FullName | Measure-Object -Line).Lines
        [PSCustomObject]@{
            File = $_.FullName
            Lines = $lineCount
        }
    } | Sort-Object Lines -Descending | Select-Object -First 10

# Dependency Analysis
mvn dependency:tree | Select-String "compile|runtime" | Measure-Object | Select-
Object -ExpandProperty Count
```

Phase 2: Code-Quality-Check (10 Min)

🐧 Linux/macOS (Bash):

```
# Find potential problem patterns
grep -r "System.out.println" src/main/java/ | wc -l
grep -r "e.printStackTrace" src/main/java/ | wc -l
grep -r "TODO\|FIXME\|HACK" src/main/java/ | wc -l

# Find God Classes (>1000 lines)
```

```
find . -name "*.java" -not -path "*/target/*" -exec wc -l {} \; | awk '$1 > 1000
{print $2 " (" $1 " lines)"}'

# Find potential SQL injection risks
grep -r "\"SELECT\|\"UPDATE\|\"DELETE\|\"INSERT" src/main/java/ | grep -v
"PreparedStatement" | wc -l
```

🪟 Windows (PowerShell):

```
# Find potential problem patterns
(Get-ChildItem -Recurse -Path "src\main\java" -Include "*.java" |
    Select-String "System.out.println").Count

(Get-ChildItem -Recurse -Path "src\main\java" -Include "*.java" |
    Select-String "e.printStackTrace").Count

(Get-ChildItem -Recurse -Path "src\main\java" -Include "*.java" |
    Select-String "TODO|FIXME|HACK").Count

# Find God Classes (>1000 lines)
Get-ChildItem -Recurse -Path "src\main\java" -Include "*.java" |
    ForEach-Object {
        $lineCount = (Get-Content $_.FullName | Measure-Object -Line).Lines
        if ($lineCount -gt 1000) {
            "$($_.FullName) ($lineCount lines)"
        }
    }

# Find potential SQL injection risks
$sqlPatterns = Get-ChildItem -Recurse -Path "src\main\java" -Include "*.java" |
    Select-String '"SELECT|"UPDATE|"DELETE|"INSERT'
$preparedStatements = Get-ChildItem -Recurse -Path "src\main\java" -Include
"*.java" |
    Select-String "PreparedStatement"
$potentialRisks = $sqlPatterns.Count - $preparedStatements.Count
Write-Output "Potential SQL injection risks: $potentialRisks"
```

Phase 3: Test-Coverage-Assessment (10 Min)

🐧 Linux/macOS (Bash):

```
# Run existing tests and check coverage
mvn clean test jacoco:report

# Count test files vs production files
echo "Production files: $(find src/main/java -name "*.java" | wc -l)"
echo "Test files: $(find src/test/java -name "*.java" | wc -l)"

# Check for integration tests
find . -name "*IT.java" -o -name "*IntegrationTest.java" | wc -l
```

🪟 Windows (PowerShell):

```
# Run existing tests and check coverage
mvn clean test jacoco:report

# Count test files vs production files
$prodFiles = (Get-ChildItem -Recurse -Path "src\main\java" -Include
"*.java").Count
$testFiles = (Get-ChildItem -Recurse -Path "src\test\java" -Include
"*.java").Count
Write-Output "Production files: $prodFiles"
```

```
Write-Output "Test files: $testFiles"

# Check for integration tests
$integrationTests = (Get-ChildItem -Recurse -Include "*IT.java",
"*IntegrationTest.java").Count
Write-Output "Integration tests: $integrationTests"
```

📊 Complete Assessment Scorecard

1. Code-Basis-Assessment

Kriterium	Gut (3)	Mittel (2)	Schlecht (1)	Punkte
Lines of Code	< 50k	50k-200k	> 200k	___
Durchschnittliche Klassengröße	< 200 LOC	200-500 LOC	> 500 LOC	___
Anzahl God Classes (>1000 LOC)	0	1-3	> 3	___
Package-Struktur	Logisch organisiert	Gemischt	Chaotisch	___
Code-Dubletten	< 5%	5-15%	> 15%	___

Code-Basis Score: ___/15

2. Test-Coverage-Assessment

Kriterium	Gut (3)	Mittel (2)	Schlecht (1)	Punkte
Unit Test Coverage	> 80%	50-80%	< 50%	___
Test/Code Ratio	> 0.5	0.2-0.5	< 0.2	___
Integration Tests	Vorhanden	Teilweise	Keine	___
Test-Qualität	Aussagekräftig	Gemischt	Dummy-Tests	___
Test-Geschwindigkeit	< 5 Min	5-15 Min	> 15 Min	___

Test Score: ___/15

3. Dependency-Assessment

Kriterium	Gut (3)	Mittel (2)	Schlecht (1)	Punkte
Anzahl Dependencies	< 20	20-50	> 50	___
Veraltete Dependencies	0	1-3	> 3	___
Security Vulnerabilities	0	1-2	> 2	___
Circular Dependencies	Keine	1-2	> 2	___
Framework-Kopplung	Lose gekoppelt	Mittel gekoppelt	Stark gekoppelt	___

Dependency Score: ___/15

4. Business-Logic-Assessment

Kriterium	Gut (3)	Mittel (2)	Schlecht (1)	Punkte
Domain Model Clarity	Klar erkennbar	Teilweise	Unklar	___
Business Rules Location	Zentralisiert	Verteilt	Überall	___
Data Access Patterns	Konsistent	Gemischt	Chaotisch	___
Error Handling	Strukturiert	Teilweise	Inkonsistent	___
Logging Strategy	Professionell	Basic	System.out	___

Business Logic Score: ___/15

Gesamt-Score: ___/60

🎯 Risiko-Matrix-Bewertung

Score-Interpretation:

- **50-60 Punkte**: ✅ **Niedrig-Risiko** - Modernisierung gut machbar
- **35-49 Punkte**: ⚠️ **Mittel-Risiko** - Vorsichtige Modernisierung nötig
- **20-34 Punkte**: 🚨 **Hoch-Risiko** - Charakterisierungstests essentiell
- **< 20 Punkte**: ☠️ **Kritisch** - Rewrite erwägen

Zusätzliche Risiko-Faktoren:

Faktor	Risiko-Multiplikator
Mission-Critical System	x 1.5
Keine Original-Entwickler verfügbar	x 1.3
Komplexe Business-Regeln	x 1.2
Multiple Teams betroffen	x 1.4
Regulatory Compliance	x 1.3

Finaler Risiko-Score: ___ (Basis-Score × Multiplikatoren)

📋 Detaillierte Checklisten

Pre-Migration-Checklist

📁 Code-Inventar

- [ ] Gesamte Lines of Code ermittelt
- [ ] Top 10 größte Klassen identifiziert
- [ ] Package-Dependencies visualisiert
- [ ] Code-Dubletten-Report erstellt
- [ ] Dead Code identifiziert

🧪 Test-Inventar

- [ ] Aktuelle Test-Coverage gemessen
- [ ] Flaky Tests identifiziert
- [ ] Test-Execution-Zeit dokumentiert
- [ ] Integration Test Gaps identifiziert
- [ ] Performance Test Baseline erstellt

📦 Dependency-Inventar

- [ ] Dependency-Tree analysiert
- [ ] Security Vulnerabilities gescannt

- [ ] License-Compliance geprüft
- [ ] Veraltete Dependencies identifiziert
- [ ] Transitive Dependencies verstanden

💼 Business-Inventar

- [ ] Stakeholder identifiziert
- [ ] Business-Rules dokumentiert
- [ ] Critical User Journeys definiert
- [ ] Performance-Requirements erfasst
- [ ] Downtime-Toleranz geklärt

Charakterisierung-Test-Strategie

🎯 High-Priority Kandidaten für Charakterisierung

```
Bewertungskriterien (je höher, desto wichtiger):
1. Business-Kritikalität (1-5)
2. Code-Komplexität (Cyclomatic Complexity)
3. Change-Frequency (Git-Commits/Monat)
4. Test-Coverage (je niedriger, desto höher Priorität)
5. Bug-History (Anzahl Bugs in letzten 12 Monaten)

Formel: Priorität = (Business-Kritikalität × 3) + (Komplexität × 2) + (Change-
Frequency × 2) + ((100 - Coverage) × 0.05) + (Bug-Count × 1)
```

✅ Charakterisierungstest-Checkliste

- [ ] **Input-Output-Mapping**: Dokumentiere alle Ein-/Ausgabe-Kombinationen
- [ ] **Edge-Cases**: Teste Grenzwerte und Sonderfälle
- [ ] **Error-Paths**: Dokumentiere Fehlerverhalten
- [ ] **Side-Effects**: Erfasse alle Seiteneffekte (DB, Files, Logs)
- [ ] **Performance-Baseline**: Mess aktuelle Performance
- [ ] **Integration-Points**: Teste externe Dependencies

🧪 Golden Master Testing Setup

```
// Template für Golden Master Tests
@Test
void characterizeComplexBusinessLogic() {
    // Arrange - Setup mit realistischen Produktionsdaten
    ComplexBusinessService service = new ComplexBusinessService();
    List<BusinessInput> inputs = loadProductionLikeData();

    // Act - Führe komplexe Operation aus
    List<BusinessOutput> outputs = inputs.stream()
        .map(service::processBusinessLogic)
        .collect(toList());

    // Assert - Vergleiche mit Golden Master
    ApprovalTests.verifyAsJson(outputs);
}
```

📈 Migration-Roadmap-Template

Phase 1: Foundation (2-4 Wochen)

Ziel: Sicherheitsnetz aufbauen

Woche 1-2: Charakterisierungstests

- [ ] Top 5 kritische Klassen charakterisieren
- [ ] Golden Master Tests für komplexe Algorithmen
- [ ] End-to-End Tests für kritische User Journeys
- [ ] Performance-Baselines etablieren

Woche 3-4: Test-Infrastructure

- [ ] CI/CD-Pipeline für Tests etablieren
- [ ] Test-Data-Management aufbauen
- [ ] Mutation Testing einführen
- [ ] Test-Coverage-Monitoring aktivieren

Phase 2: Isolation (3-6 Wochen)

Ziel: Code isolierbar und testbar machen

Dependency-Injection einführen

- [ ] Service-Layer isolieren
- [ ] Repository-Pattern implementieren
- [ ] Configuration externalisieren
- [ ] Feature-Flags vorbereiten

Seams einführen

- [ ] Interface-Abstraktionen schaffen
- [ ] Testable Boundaries definieren
- [ ] Mock-Points identifizieren
- [ ] Integration-Test-Doubles vorbereiten

Phase 3: Modernization (4-8 Wochen)

Ziel: Moderne Patterns einführen

Code-Qualität verbessern

- [ ] SOLID-Principles anwenden
- [ ] Design-Patterns implementieren
- [ ] Error-Handling standardisieren
- [ ] Logging-Strategy implementieren

Technology-Upgrades

- [ ] Java-Version upgraden
- [ ] Framework-Updates durchführen
- [ ] Security-Patches anwenden

- [ ] Performance-Optimierungen

Phase 4: Validation (2-3 Wochen)

Ziel: Qualität sicherstellen

Quality-Gates definieren

- [ ] Code-Coverage > 80%
- [ ] Mutation Score > 85%
- [ ] Performance-Regression < 5%
- [ ] Security-Scan clean

User-Acceptance

- [ ] Stakeholder-Reviews
- [ ] User-Testing durchführen
- [ ] Production-Monitoring aktivieren
- [ ] Rollback-Strategie testen

🛠️ Tools und Automation

Code-Analysis-Tools

🐧 Linux/macOS (Bash):

```
# SonarQube für Code-Quality
docker run -d --name sonarqube -p 9000:9000 sonarqube:latest

# PMD für Code-Smells
mvn pmd:pmd pmd:cpd

# SpotBugs für Bug-Detection
mvn com.github.spotbugs:spotbugs-maven-plugin:spotbugs

# ArchUnit für Architecture-Testing
# Siehe Beispiel unten
```

🪟 Windows (PowerShell):

```
# SonarQube für Code-Quality (Docker Desktop erforderlich)
docker run -d --name sonarqube -p 9000:9000 sonarqube:latest

# PMD für Code-Smells
mvn pmd:pmd pmd:cpd

# SpotBugs für Bug-Detection
mvn com.github.spotbugs:spotbugs-maven-plugin:spotbugs

# Windows-specific: Code-Analyse mit PowerShell
# Komplexitäts-Analyse
Get-ChildItem -Recurse -Include "*.java" |
    ForEach-Object {
        $content = Get-Content $_.FullName -Raw
        $methodCount = ([regex]::Matches($content, '\b(public|private|
protected)\s+\w+\s+\w+\s*\(')).Count
```

```
        $classCount = ([regex]::Matches($content, '\b(class|interface|enum)\s+\
w+')).Count
        [PSCustomObject]@{
            File = $_.Name
            Methods = $methodCount
            Classes = $classCount
            Lines = (Get-Content $_.FullName | Measure-Object -Line).Lines
        }
    } | Sort-Object Lines -Descending | Format-Table -AutoSize
```

Architecture-Testing mit ArchUnit

```
@ArchTest
static final ArchRule services_should_not_access_controllers =
    noClasses()
        .that().resideInAPackage("..service..")
        .should().accessClassesThat().resideInAPackage("..controller..");

@ArchTest
static final ArchRule repositories_should_not_use_services =
    noClasses()
        .that().resideInAPackage("..repository..")
        .should().accessClassesThat().resideInAPackage("..service..");
```

Security-Assessment-Tools

🐧 Linux/macOS (Bash):

```
# OWASP Dependency Check
mvn org.owasp:dependency-check-maven:check

# Snyk für Vulnerability Scanning (Node.js erforderlich)
npm install -g snyk
snyk test

# Git-Secrets für Secret-Detection
git-secrets --scan
```

🪟 Windows (PowerShell):

```
# OWASP Dependency Check
mvn org.owasp:dependency-check-maven:check

# Snyk für Vulnerability Scanning (Node.js erforderlich)
npm install -g snyk
snyk test

# Git-Secrets für Secret-Detection (Windows-Installation erforderlich)
# git-secrets --scan

# Windows-alternative: PowerShell-basierte Secret-Detection
Get-ChildItem -Recurse -Include "*.java", "*.properties", "*.yml", "*.yaml" |
    Select-String -Pattern "(password|secret|key|token)\s*[=:]\s*['\"]?[^'\"\s]+
['\"]?" -AllMatches |
    ForEach-Object {
        Write-Warning "Potential secret found in $($_.Filename):$($_.LineNumber)
- $($_.Line.Trim())"
    }

# Check for hardcoded IPs and URLs
Get-ChildItem -Recurse -Include "*.java" |
```

```
    Select-String -Pattern "https?://|(\d{1,3}\.){3}\d{1,3}" |
    ForEach-Object {
        Write-Output "URL/IP found in $($_.Filename):$($_.LineNumber) - $
($_.Line.Trim())"
    }
```

📋 Decision-Framework

Go/No-Go Entscheidungsmatrix

✅ GO für Modernisierung wenn:

- [ ] Score ≥ 35 UND Risiko-Multiplikator < 2.0
- [ ] Business-Value klar definiert
- [ ] Team committed für ≥ 6 Monate
- [ ] Budget für 1.5× der geschätzten Zeit
- [ ] Rollback-Strategie vorhanden

⚠️ CONDITIONAL GO wenn:

- [ ] Score 25-34 UND starke Business-Begründung
- [ ] Phased Approach möglich
- [ ] Pilot-Phase erfolgreich
- [ ] Externe Expertise verfügbar

❌ NO-GO wenn:

- [ ] Score < 25 ODER Risiko-Multiplikator > 2.5
- [ ] Keine klaren Service-Boundaries
- [ ] Team-Turnover > 50% erwartet
- [ ] Mission-critical ohne Rollback-Option

Alternative-Strategien

🔄 Strangler Fig Pattern

Wann: Bei Score 20-35 mit klaren Boundaries

- Neue Features in neuem System
- Schrittweise Migration alter Features
- Parallel-Run für kritische Komponenten

🏗️ Big Bang Rewrite

Wann: Bei Score < 20 UND nicht business-critical

- Kompletter Neubau mit modernen Patterns
- Legacy als Black-Box betrachten
- Umfangreiche Characterization Tests

🔧 Incremental Refactoring

Wann: Bei Score > 35 mit guter Test-Coverage

- Schrittweise Verbesserung bestehender Code-Base
- Boy Scout Rule: "Leave it better than you found it"
- Kontinuierliche kleine Verbesserungen

📊 Erfolgs-Metriken

Technical Metrics

```
Vor Migration → Nach Migration:
- Code Coverage: ___% → Ziel: >80%
- Mutation Score: ___% → Ziel: >85%
- Build Time: ___min → Ziel: <10min
- Test Execution: ___min → Ziel: <5min
- Cyclomatic Complexity: ___ → Ziel: <10 avg
- Technical Debt: ___hrs → Ziel: <100hrs
```

Business Metrics

```
- Feature Delivery Speed: ___% improvement
- Bug Rate: ___% reduction
- Production Incidents: ___% reduction
- Developer Satisfaction: ___/10
- Time to Market: ___% improvement
- Maintenance Cost: ___% reduction
```

Quality Gates für Deployment

```
Alle Kriterien müssen erfüllt sein:
☐ All Charakterisierung Tests pass
☐ Code Coverage ≥ 80%
☐ Mutation Score ≥ 85%
☐ Performance Regression < 5%
☐ Security Scan: 0 High/Critical issues
☐ Architecture Tests pass
☐ Stakeholder Acceptance erhalten
☐ Rollback-Plan tested
```

🎯 Success Stories Template

Template für Lessons Learned

```
## Legacy Migration: [Project Name]

### Projekt-Kontext:
- **System**: [Beschreibung]
- **Alter**: [Jahre]
- **LOC**: [Anzahl]
- **Team**: [Größe]
- **Timeline**: [Wochen]
```

```
### Assessment-Ergebnisse:
- **Initial Score**: ___/60
- **Risiko-Level**: [Niedrig/Mittel/Hoch]
- **Hauptprobleme**: [Top 3]

### Migration-Strategie:
- **Gewählter Ansatz**: [Strangler/Incremental/Rewrite]
- **Phasen**: [Anzahl und Dauer]
- **Test-Strategie**: [Charakterisierung/Unit/Integration]

### Ergebnisse:
- **Timeline**: Geplant vs. Actual
- **Qualitäts-Verbesserung**: [Metriken]
- **Business-Impact**: [Messbarer Nutzen]

### Lessons Learned:
#### ✅ Was gut funktioniert hat:
1. [Erfolgreiche Strategie 1]
2. [Erfolgreiche Strategie 2]
3. [Erfolgreiche Strategie 3]

#### ❌ Was wir anders machen würden:
1. [Verbesserungsbereich 1]
2. [Verbesserungsbereich 2]
3. [Verbesserungsbereich 3]

#### 💡 Empfehlungen für ähnliche Projekte:
1. [Praktischer Tipp 1]
2. [Praktischer Tipp 2]
3. [Praktischer Tipp 3]
```

🚀 Nutze dieses Assessment-Kit als systematische Grundlage für jede Legacy-Modernisierung!

💡 Denk daran: Gründliche Vorbereitung ist der Schlüssel zum Erfolg bei Legacy-Migrationen!

Dieses Assessment-Kit basiert auf realen Projekterfahrungen und wird kontinuierlich erweitert.