FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit

Analyse der Wirtschaftlichkeit von „Large-Scale Refactoring“ Tools am Beispiel von OpenRewrite

Autor:
Gary Grandt, 3526976

Betreuer:
Prof. Dr. rer. nat. Philipp Rohde
M. Sc. Lukas Kirchhart

Aachen, 8. Januar 2024

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Analyse der Wirtschaftlichkeit von „Large-Scale Refactoring" Tools am Beispiel von OpenRewrite

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Gary Grandt

Aachen, den 08.01.2024

G. Grandt

Unterschrift der Studentin / des Studenten

Zusammenfassung

Die MAGMA Gießereitechnologie GmbH hat, wie viele andere Firmen, eine große Menge Bestandscode. Diesen auf dem aktuellen Stand der Technik zu halten stellt eine große Herausforderung dar. Insbesondere die Migration zu einer neueren Framework-Version, wie bei der Migration von JUnit 4 nach JUnit 5, bringt viele API-Änderungen mit sich. Diese sind im Einzelnen nicht schwer umzusetzen, werden jedoch aufgrund der Masse an Sourcecode, der geändert werden muss, zur Herausforderung.

In dieser Seminararbeit wird untersucht, ob Large-Scale Refactoring-Tools eine kosteneffiziente Lösung zur Instandhaltung großer Codebasen bieten können. Zur Bewertung des wirtschaftlichen Mehrwerts werden verschiedene Kriterien festgelegt. Anhand dieser Kriterien wird ein Tool zur weiteren Evaluation ausgewählt. Dieses wird verwendet, um die bestehenden JUnit 4 Tests nach JUnit 5 zu migrieren. Anschließend werden die Ergebnisse anhand der zuvor festgelegten Kriterien ausgewertet.

Bei der Vorabauswahl wurde OpenRewrite zur weiteren Evaluation ausgewählt. Der praktische Einsatz von OpenRewrite war erfolgreich und hat gezeigt, dass die Kriterien zur Bewertung der Wirtschaftlichkeit erfüllt werden. Jedoch gibt es auch Limitierungen, wie eine fehlende Validierung des migrierten Sourcecodes oder, dass OpenRewrite nur auf ganze Java Projekte und nicht auf einzelne Klassen oder Packages ausgeführt werden kann. Abschließend ist davon auszugehen, dass OpenRewrite einen wirtschaftlichen Mehrwert für die Entwicklung von MAGMASOFT® bieten kann.

Inhaltsverzeichnis

1 Einleitung und Motivation

Bei der MAGMA Gießereitechnologie GmbH wird seit 1988 mit MAGMASOFT® [Gmb] eine Softwarelösung für Gießprozess-Simulation entwickelt. Daher existiert eine dementsprechend große Codebasis. So gibt es circa 2,4 Millionen Zeilen Javacode. Davon werden ungefähr 600.000 Zeilen durch einen Codegenerator erzeugt. Dazu kommen weitere 250.000 Zeilen JUnit-Tests, welche auf über 2.000 JUnit-Testklassen aufgeteilt sind. Im Modellteam arbeiten sechs Softwareentwickler. Dies führt zu einem hohen Verhältnis von Sourcecode zu Entwicklern. So kommen auf einen einzelnen Entwickler:

- 250.000 Zeilen Sourcecode
- 100.000 Zeilen generierter Sourcecode
- 40.000 Zeilen JUnit-Tests

Diese umfangreiche Codebasis bei entsprechendem Verhältnis zu Entwicklern bleibt nicht ohne Konsequenzen. So werden neue Features häufig in das System integriert, ohne dass Zeit für Refactoring oder das Verbessern der zugrundeliegenden Architektur bleibt. Dies führt zu einem Anstieg der technischen Schulden und ist langfristig nicht wünschenswert. Eine Herausforderung stellt zusätzlich die Instandhaltung einer solch großen Codebasis dar. Diese verwendet hauptsächlich Java und nutzt die Frameworks JUnit [Gar17] und Mockito [Kac19] im Bereich der Tests. Da sowohl Java als auch die verwendeten Frameworks weiterentwickelt werden, ist es aufwändig, diese auf dem aktuellen Stand der Technik zu halten. Beispielhaft hierfür ist der Wechsel von JUnit 4 [JUn23a] nach JUnit 5 [JUn23b]. Diese Umstellung ist in den einzelnen Klassen leicht umzusetzen. So muss bei der Migration von JUnit 4 nach JUnit 5 und den daraus folgenden API-Änderungen größtenteils nur der Import und die Position der Fehlermeldung geändert werden. Problematisch ist jedoch die große Anzahl an Klassen, die migriert werden müssen.

```
@Test
public void junit4() {
    assertEquals(msg, expected, actual);
}
```

```
@Test
public void junit5() {
    assertEquals(expected, actual, msg);
}
```

Abbildung 1.1: Änderungen von JUnit 4 nach JUnit 5

Ähnlich verhält es sich mit der Migration zu neueren Java Versionen. Java ist zwar rückwärts kompatibel, das bloße Erhöhen der Java Version bringt jedoch nicht zwangsläufig etwaige Performance Verbesserungen mit sich. Ein Beispiel dafür ist die Änderungen bei der Erzeugung von Longs:

```
new Long(42);       // Java 1.8
Long.valueOf(42);   // Java 11
```

Abbildung 1.2: Änderung bei der Erzeugung von Java Longs

Wird am Codegenerator gearbeitet, müssen häufig vergleichbare Anpassungen vorgenommen werden. Die bestehenden Refactoring-Tools bieten bei API-Änderungen dieser Art nur wenig bis gar keine Unterstützung. Deshalb bedeutet die Implementierung dieser Änderungen für den Entwickler meist viel Handarbeit. Dies ist zeitaufwendig und daher kostenintensiv, aber auch

fehleranfällig. Aufgrund dessen ist es wünschenswert, diese oft einfachen, aber häufig anfallenden Schritte zu automatisieren. In dieser Seminararbeit wird untersucht, ob OpenRewrite [Ope23b] eine kosteneffiziente Lösung zur Automatisierung bietet. Hierzu wird zuerst analysiert, welche Anforderungen die MAGMA Gießereitechnologie GmbH an OpenRewrite hat, damit sich der Einsatz des Tools lohnt. Anschließend wird OpenRewrite zur Migration von JUnit verwendet und die Ergebnisse anhand von definierten Kriterien bewertet.

2 Stand der Technik

Einige der in dieser Arbeit verwendeten Begriffe wie *Refactoring* oder *Migration* können unterschiedlich interpretiert werden. Deshalb wird in diesem Kapitel erläutert, wie diese Begriff, in dieser Arbeit verstanden werden. Im zweiten Teil des Kapitels (2.2) wird die grundlegende Funktionsweise von OpenRewrite beschrieben. Da sich für die weitere Evaluation von OpenRewrite entschieden wurde, wird auf eine ausführliche Beschreibung der anderen Kandidaten verzichtet.

2.1 Definition von Migration und Refactoring

Unter dem Begriff Refactoring versteht man die Restrukturierung von Sourcecode. Zum besseren Verständnis kann die Vorstellung einer schwarzen Box hilfreich sein. Diese hat eine interne Verdrahtung und Schnittstellen nach außen. Beim Refactoring wird diese Box nun intern neu verdrahtet, wobei das Verhalten nach außen hin unverändert bleibt. So definiert Martin Fowler in seinem Buch "Refactoring: Improving the Design of Existing Code" [Fow19] dieses wie folgt:

Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

Das Refactoring oder allgemeiner das Ändern von Software ist aufwändig und verursacht somit auch Kosten. Daher ist es sinnvoll sich zu überlegen, wann dieser Aufwand betrieben werden sollte. Michael C. Feather definiert in seinem Buch [Fea04] "Working effectively with legacy code" vier Gründe, warum man Software ändern sollte: *1. Adding a feature, 2. Fixing a bug, 3. Improving the design, 4. Optimizing resource usage* . Refactoring kann für die ersten beiden Punkte nützlich sein. Der Fokus liegt jedoch darauf, das Design der Software zu verbessern, Redundanzen zu entfernen und somit den Code verständlicher und wartbarer zu machen.

Insbesondere größere Refaktorisierungen benötigen eine klare Strategie, um erfolgreich zu sein. Um dies zu erreichen, stellt Feather in seinem Buch [Fea04] einen fundierten Algorithmus vor:

1. Identify change points.
2. Find test points.
3. Break dependencies.
4. Write tests.
5. Make changes and refactor.

In anderen Worten, es wird zuerst festgestellt, was geändert werden soll. Anschließend werden Testpunkte identifiziert. Im dritten Schritt werden einzelne Einheiten aus dem Sourcecode extrahiert, um so testbare Einheiten (Units) zu erhalten. Darauffolgend können Tests für die einzelnen Einheiten geschrieben werden (Unit-Tests). Nach diesen Schritten ist sichergestellt, dass nach dem Refactoring noch dasselbe Verhalten gegeben ist. Sollte bereits testgetrieben entwickelt werden (test driven development [Bec02]) oder bereits Tests bestehen, können häufig die ersten Punkte einfach übersprungen werden. In diesem Fall ist es aber immer noch hilfreich, sich der notwendigen Bedingungen bewusst zu sein, bevor im letzten Schritt die eigentlichen Änderungen durchgeführt

werden. Refactoring lässt sich grundlegend in vier Kategorien unterteilen, wobei die Übergänge oft fließend sind:

1. Restrukturierung von Sourcecode durch Extrahieren von Methoden oder dem Umstrukturieren in Klassen, um kleinere und besser benannte Teile zu erhalten
2. Entfernen von redundantem Code
3. Verbessern des Software-Designs, durch zum Beispiel das Implementieren von Design-Pattern
4. API-Änderungen, zum Beispiel das Extrahieren vieler Übergabeparameter in eine Datenklasse

Als Migration wird im Kontext der Softwareentwicklung meist ein Technologiewechsel bezeichnet. Dies kann den Wechsel zu einer höheren Framework-Version darstellen, zum Beispiel von JUnit 4 nach JUnit 5 [Gar17], oder den Wechsel zu einem anderen Framework, wie zum Beispiel von JUnit nach AssertJ [SBPG18]. Anders als bei der vierten Kategorie von Refactoring wird hier jedoch das externe Verhalten geändert. Somit bricht die Migration die Refactoring-Definition von Martin Fowler [Fow19]. Daher ist unter dem Begriff Migration in dieser Seminararbeit, eine Veränderung des Sourcecodes zu verstehen, welches das Verhalten ändert. Ein Beispiel hierfür ist der Wechsel von JUnit 4 nach JUnit 5. Der Begriff Refactoring hingegen folgt der Definition von Martin Fowler.

Bei der Entwicklung von MAGMASOFT® werden hauptsächlich die in der Eclipse IDE [Fou] integrierten Refactoring-Tools eingesetzt. Diese bieten leider kaum Unterstützung bei der Migration von Frameworks, weswegen viele API-Änderungen von Hand umgesetzt werden müssen. Dies ist sehr zeitaufwendig und somit kostenintensiv. Dazu kommt, dass ein Teil des Modells von MAGMASOFT® durch einen Codegenerator erzeugt wird. Änderungen an diesem sind für die IDE vergleichbar mit einer API-Änderung. Dies kann dazu führen, dass nach einer Änderung am Codegenerator viele Stellen von Hand angepasst werden müssen. Das im nächsten Unterkapitel (2.2) beschriebene Tool OpenRewrite verspricht zum einen, die bestehenden Frameworks kosteneffizient auf eine neue Version migrieren zu können. Zum anderen kann OpenRewrite durch die Möglichkeit, eigene Routinen zu schreiben, auch eine Lösung für die Herausforderung der durch den Codegenerator verursachten API-Änderungen bieten.

2.2 OpenRewrite

OpenRewrite ist ein Open-Source-Tool, um automatisiert Änderungen am Sourcecode durchzuführen. Es wurde ursprünglich von Jonathan Schneider bei Netflix entwickelt, um von einem selbst entwickelten Logging-Framework zu SLF4J [Che17] zu wechseln. Zusammen mit Olga Kundzich gründete er die Firma Moderne [Mod], diese entwickelt OpenRewrite und vermarktet die Moderne CLI. Mit der Moderne CLI ist es möglich, die Laufzeit von OpenRewrite zu verkürzen.

OpenRewrite selbst arbeitet mit sogenannten Rezepten (Recipes), diese sind quelloffen und können von jedem weiterentwickelt werden. OpenRewrite wird in ein Maven [Var19a] oder Gradle [Mus14] Build-File eingebunden [Ope23h]. Dort werden auch die Rezepte als Abhängigkeit (dependencies) angegeben. Dies ermöglicht es ebenfalls, Refaktorisierungsprozesse automatisiert in einer CI-Pipeline [vM23] laufen zu lassen. Es gibt beispielsweise Rezepte für:

- Die Migration von Java Versionen, zum Beispiel von Java 11 nach Java 17
- Die Migration von JUnit 4 nach JUnit 5
- Die Migration von Spring Boot
- Die Migration von Mockito

Weitere Rezepte sind zu finden unter `https://docs.openrewrite.org/recipes`.

Darüber hinaus gibt es auch Rezepte für das Optimieren von Suchalgorithmen oder das Finden und Beheben von bekannten Sicherheitslücken. Mittlerweile unterstützt OpenRewrite weitere Sprachen wie Kotlin, Python oder XML. Aufgrund des Open-Source Ansatzes kann jeder Entwickler auch eigene Rezepte schreiben, die seine individuellen Bedürfnisse erfüllen.

2.2.1 Funktionsweise von OpenRewrite

OpenRewrite baut intern eine Baumstruktur auf, welche den Sourcecode repräsentiert. Auf diese Baumstruktur werden die Rezepte angewendet und anschließend wird aus der veränderten Struktur wieder Sourcecode generiert. In diesem Abschnitt wird zuerst die herkömmliche Technik für solche Baumstrukturen erklärt. Anschließend wird auf die Limitierung dieser eingegangen und abschließend die von OpenRewrite gewählte Lösung vorgestellt.

Ein Abstract Syntax Tree (Abstrakter Syntaxbaum) oder kurz AST [Noo85] genannt, ist eine Baumstruktur, welche die syntaktische Struktur von Sourcecode darstellt. Diese wird häufig von Compilern oder Interpretern verwendet, um den Sourcecode zu verarbeiten. Innerhalb des Baumes repräsentieren Knoten die einzelnen Syntax-Elemente und Kanten die Beziehungen zwischen den Elementen. Das nachfolgende Bild 2.1 zeigt ein Beispiel für solch eine Baumstruktur.

Abbildung 2.1: Beispiel eines AST

Da der AST ursprünglich für Compiler-Anwendungen entwickelt wurde, werden keine nicht-syntaktischen Informationen gespeichert. Zudem speichert der AST die Information auf abstrakter Ebene, was zu einer fehlenden Typsicherheit führt. Da OpenRewrite nach der Anwendungen eines Rezeptes auf den Baum aus diesen wieder Sourcecode generiert, ist beides problematisch.

Als Lösung für die Limitierung des AST verwendet OpenRewrite einen Lossless Semantic Tree (Verlustfreier semantischer Baum) oder kurz LST [Ope23c] genannt. Dieser ist eine Weiterentwicklung des AST und löst dessen Limitierungen bei OpenRewrite. Um dies zu erreichen, wird:

- Das Format erhalten.
  Der LST speichert Informationen über Leerzeichen, sodass nach der Deserialisierung unveränderte Klassen nicht die Formatierung verlieren. Des Weiteren passt sich neu eingefügter Sourcecode der existierenden Formatierung an.

- Die Typinformation gespeichert.
  Eine Limitierung des AST ist, dass keine exakten Typinformationen gespeichert werden. So ist nicht rekonstruierbar, ob bei der Operation `logger.info(message)` SLF4J oder Logback verwendet wird. Daher speichert der LST den exakten Typ eines Objektes.

Außerdem kann der von OpenRewrite verwendete LST vollständig serialisiert und in einer JSON-Datei gespeichert werden. So wird es ermöglicht, den LST vorab zu generieren. Dies ist insbesondere bei der Anwendungen auf große Sourcecode-Basen wichtig, da sonst sehr viel Arbeitsspeicher benötigt werden kann.

Um die Anwendung von Rezepten besser verstehen zu können, ist es hilfreich, sich den Lifecycle eines LST anzuschauen. Dieser ist in der Dokumentation von OpenRewrite [Ope23c] wie folgt angegeben:

1. OpenRewrite erstellt aus dem Sourcecode einen LST, der im Arbeitsspeicher abgelegt wird. Dieser spiegelt den aktuellen Zustand des Repositories auf der Festplatte wider.

2. Als Nächstes werden die vom Rezept vorgegeben Änderungen am LST durchgeführt.

3. Anschließend wird aus dem LST wieder Sourcecode generiert. Dieser wird verwendet, um die Dateien mit Änderungen zu überschreiben.

4. Nach dem Schreiben aller Änderungen wird der LST aus dem Arbeitsspeicher gelöscht.

Wird anschließend ein weiteres Rezept ausgeführt, startet der Zyklus von vorne und es wird ein neuer LST generiert.

2.2.2 Funktionsweise von Rezepten

In diesem Abschnitt wird die Funktionsweise und Entwicklung eigener Rezepte betrachtet [Ope23a]. Nach der Generierung des LST werden die Rezepte auf diesen angewendet. Hierzu wird das Visitor-Pattern [ES13] verwendet. Um eigene Rezepte zu erstellen, wird eine Java-Klasse implementiert, welche von der org.openrewrite.Recipe-Klasse erbt. In dieser werden für gewöhnlich die grundlegenden Eigenschaften des Rezeptes festgelegt, wie der Name, eine Beschreibung und die verwendeten Visitor. Als Nächstes wird die Visitor-Klasse implementiert. Dies geht als eigenständige Klasse oder als anonyme innere Klasse. Diese muss den org.openrewrite.java.JavaIsoVisitor implementieren. Innerhalb der Visitor-Klasse kann auf verschiedene Callback-Methoden zugegriffen werden. So kann über die visitCompilationUnit(...)-Methode auf die Imports der Klasse zugegriffen werden und zum Beispiel bei der Migration von JUnit *import org.junit.Test;* zu *import org.junit.jupiter.api.Test;* geändert werden. Die nachfolgende Abbildung 2.2 zeigt das in 4.2.3 verwendet Rezept zum Migrieren der MAGMA-Rules.

```
public class ConvertMAGMARules extends Recipe {
  @Override
  public String getDisplayName() {
    return "Converts MAGMA JUnit Rules";
  }

  @Override
  public String getDescription() {
    return "Changes MAGMA JUnit Rules to register Extention.";
  }

  @Override
  public TreeVisitor<?, ExecutionContext> getVisitor() {
    return new JavaIsoVisitor<ExecutionContext>() {

      private static final String JUNIT_RULE = "org.junit.Rule";
      private static final String JUNIT_DEFAULT_RULE = "org.junit.rules";
      private static final String JUNIT_EXTENSION = "org.junit.jupiter."
                                  + "api.extension.RegisterExtension";

      private boolean convertRules = false;

      @Override
      public CompilationUnit visitCompilationUnit(CompilationUnit cu, ExecutionContext p) {
        convertRules = checkImports(cu.getImports());

        return super.visitCompilationUnit(cu, p);
      }

      private boolean checkImports(List<Import> imports) {
        boolean hasJUnitRule = false;
        boolean hasNoDefaultRule = true;

        for (Import currentImport : imports) {
          String typeName = currentImport.getTypeName();

          if (typeName.equals(JUNIT_RULE)) {
            hasJUnitRule = true;
          } else if (typeName.contains(JUNIT_DEFAULT_RULE)) {
            hasNoDefaultRule = false;
          }
        }

        return hasJUnitRule && hasNoDefaultRule;
      }

      @Override
      public Annotation visitAnnotation(Annotation annotation, ExecutionContext p) {
        if (convertRules) {
          doAfterVisit(new ChangeType(JUNIT_RULE, JUNIT_EXTENSION, true).getVisitor());
        }
        return super.visitAnnotation(annotation, p);
      }
    };
  }
}
```

Abbildung 2.2: Rezept zum migrieren der MAGMA-Rules

Zum Testen von Rezepten wird eine Klasse erstellt, die das Interface *RewriteTest* implementiert. Innerhalb der Klasse wird die *defaults()*-Methode überschrieben, um so das verwendet Rezept anzugeben. In den Testmethoden kann der Stand vor und nach Anwenden des Rezeptes angegeben werden. Im Fall, dass keine Änderungen vorgenommen werden, wird nur der Sourcecode, auf dem das Rezept angewendet wird, angegeben. Die nachfolgende Abbildung 2.3 zeigt die Klassen zum Testen des Rezeptes zum Migrieren der MAGMA-Rules 2.2.

```
class ConvertMAGMARulesTest implements RewriteTest {

  @Override
  public void defaults(RecipeSpec spec) {
    spec.parser().recipe(new ConvertMAGMARules());
  }

  @Test
  void rule() {
    rewriteRun(java("""
        import org.junit.Rule;
        import de.magmasoft.test.utils.rules.MSProductRule;

        class Test {
            @Rule
            public MSProductRule mspr = new MSProductRule(Product.MAGMA);
        }
        """,
        """
        import org.junit.jupiter.api.extension.RegisterExtension;
        import de.magmasoft.test.utils.rules.MSProductRule;

        class Test {
            @RegisterExtension
            public MSProductRule mspr = new MSProductRule(Product.MAGMA);
        }
        """));
  }

  @Test
  void exceptionRule() {
    rewriteRun(java("""
          import java.lang.NumberFormatException;

          import org.junit.Rule;
          import org.junit.rules.ExpectedException;
          import de.magmasoft.test.utils.rules.MSProductRule;

          public class ExceptionRule {
              @Rule
              public MSProductRule mspr = new MSProductRule(Product.MAGMA);

              @Rule
              public ExpectedException exceptionRule = ExpectedException.none();
          }
        """));
  }
}
```

Abbildung 2.3: Klasse zum testen des Rezeptes in Abbildung 2.2

Außerdem kann aus den Callback-Methoden auf den gesamten LST zugegriffen und dieser ausgegeben werden. Zum Ausgeben des LST ist in der Dokumentation [Ope23f] von OpenRewrite der folgende Beispielcode angegeben:

```
System.out.println(TreeVisitingPrinter.printTree(getCursor()));
```

Die Abbildung 2.4 zeigt eine einfache HelloWorld-Klasse. Die Visualisierung des LST dieser Klasse ist in der nachfolgenden Abbildung 2.5 zu sehen.

```
public class HelloWorld {
    private String message = "Hello World!";

    public void printMessage() {
        System.out.println(message");
    }
}
```

Abbildung 2.4: Beispiel HelloWorld-Klasse

```
J.ClassDeclaration
|---J.Modifier | "public"
|---J.Identifier | "HelloWorld"
\---J.Block
    |---J.VariableDeclarations | "private String message = "Hello World!""
    |   |---J.Modifier | "private"
    |   |---J.Identifier | "String"
    |   \-------J.VariableDeclarations.NamedVariable | "message = "Hello World!""
    |           |---J.Identifier | "message"
    |           \-------J.Literal | ""Hello World!""
    \---J.MethodDeclaration | "MethodDeclaration{HelloWorld{return=void}}"
        |---J.Modifier | "public"
        |---J.Primitive | "void"
        |---J.Identifier | "printMessage"
        |-----------J.Empty
        \---J.Block
            \-------J.MethodInvocation | "System.out.println(message)"
                    |-------J.FieldAccess | "System.out"
                    |       |---J.Identifier | "System"
                    |       \-------J.Identifier | "out"
                    |---J.Identifier | "println"
                    \-----------J.Identifier | "message"
```

Abbildung 2.5: Ausgabe des LST

3 Anforderungsanalyse

In diesem Kapitel wird analysiert, welche Anforderungen ein Migrations-Tool erfüllen muss, damit der Einsatz einen wirtschaftlichen Mehrwert für die Entwicklung von MAGMASOFT® bietet. Ziel ist es daher, feste Kriterien herauszuarbeiten. Zu diesem Zweck wurden die vergangen zwei Migrationen von JUnit 3 nach JUnit 4 sowie von JUnit 4 nach JUnit 5 analysiert. Nach der Bestandsaufnahme werden sowohl Rahmenbedingungen als auch weitere Qualitätsmerkmale festgelegt. Im Anschluss werden mehrere Lösungsansätze präsentiert. Von diesen wird anhand der zuvor definierten Kriterien ein Ansatz zur weiteren Evaluation ausgewählt.

3.1 Analyse bestehender JUnit Migrationen

Um ein besseres Verständnis von den Problemen bei der Migration von JUnit zu erhalten, wurden die Lösungsansätze für vergangene Migration bei der MAGMA Gießereitechnologie GmbH analysiert. Die Migration von JUnit 3 nach JUnit 4 wurde 2017 größtenteils von einem Entwickler durchgeführt und hat circa einen Monat gedauert. Zu diesem Zeitpunkt gab es knapp 200 JUnit 3 Testklassen. 2021 wurde mit der Umstellung von JUnit 4 nach JUnit 5 begonnen. Jedoch war mittlerweile die Anzahl der JUnit Testklassen auf rund 800 angestiegen. Laut Schätzung hätte die Migration von JUnit 4 nach JUnit 5, einen Entwickler drei bis vier Monate durchgehend beschäftigen. Da JUnit [Gar17] das Ausführen von JUnit 4 Testcases aus einer JUnit 5 Testsuite heraus ermöglicht, wurde ein anderes Vorgehen gewählt. Die grundlegende Idee ist JUnit 5 lauffähig zu machen, sodass neue Testklassen in JUnit 5 geschrieben werden können und die bestehenden rund 800 JUnit 4 Testklassen weiter funktionsfähig bleiben. Alleine für die Migration der Testsuites und der anderen benötigten Komponenten hat ein Entwickler circa einen Monat gebraucht. Dieses Vorgehen bietet den Vorteil, dass nicht vorab ein großer Zeitaufwand ohne eine Weiterentwicklung für MAGMASOFT® geleistet werden muss. Außerdem ermöglicht es den Entwicklern einen leichteren Übergang von JUnit 4 nach JUnit 5, da die Umstellung nicht an einem Stichtag erfolgt. Zwei Jahre nach der Umstellung lassen sich jedoch auch einige Nachteile feststellen. Da häufig nicht die benötigte Zeit bleibt, um die existierenden Tests zu migrieren, werden in schätzungsweise 90% der Fälle die bestehenden JUnit 4 Tests erweitert. So kommt es, dass immer noch über 700 JUnit 4 Testklassen bestehen. Dies hat zur Folge, dass die Entwickler sowohl JUnit 4 als auch JUnit 5 können müssen. Außerdem bietet es Fehlerpotential, da das Vermischen von Frameworkversionen zu keinen Fehlermeldungen führt. Stattdessen werden Teile eines Tests nicht aufgeführt. Dieses Problem ist in den letzten Jahren so groß geworden, dass ein Tool zum Finden nicht ausgeführter Test entwickelt werden musste. Des Weiteren wird für die Entwicklung von MAGMASOFT® ein hausinternes Buildsystem verwendet, welches die Software baut, die Tests ausführt und aus den Ergebnissen eine Übersicht erstellt. Diese Buildsystem muss für beide JUnit Versionen gepflegt und angepasst werden. Abschließend lässt sich sagen, dass eine Teilmigration von JUnit kurzfristig zwar Zeit und somit Kosten spart. Langfristig verursacht diese aufgrund der genannten Nachteile jedoch höhere Entwicklungs- und Wartungskosten. Somit hat die Teilmigration laut Schätzung des Entwicklerteams bereits die benötigte Zeit zur Migration aller JUnit 4 Tests überstiegen.

3.2 Ableitung von Anforderungen

Im Folgenden wird zwischen Rahmenbedingungen und Qualitätsmerkmalen unterschieden. Rahmenbedingungen bilden hierbei Kriterien, welche eine Grundvoraussetzung sind. Als Qualitätsmerkmale werden hingegen optionale Anforderungen bezeichnet. Aus den Gesprächen mit den Mitarbeitern des Modellteams haben sich die folgenden Rahmenbedingungen ergeben:

- Kompatibilität mit Windows
- Benötigt keinen direkten Internetzugriff
- Kosten von maximal zwei Monatsgehältern eines Entwicklers pro Jahr

Diese Rahmenbedingungen ergeben sich daraus, dass alle Entwickler des Modellteams unter Windows arbeiten und in der Entwicklungsabteilung, aus Sicherheitsgründen, kein Zugriff auf das Internet möglich ist. Für die Migration der verbleibenden JUnit 4 Tests wird der zeitliche Aufwand auf zwei Monate geschätzt. Da ungewiss ist, ob das Tool nach dem Projekt weitere Verwendung findet, kann keine lange Einarbeitungszeit gerechtfertigt werden, welche zu einer Dauer von über zwei Monaten führt. Da eine zeitliche Einsparung gewünscht ist, wird die Migration als erfolgreich angesehen, wenn diese nicht länger als circa einen Monat dauert.

Zusätzlich werden die folgenden Anforderungen als Qualitätsmerkmale angesehen:

- Kompatibilität mit Linux
- Eine kurze Laufzeit
- Kann durch das bestehende Buildsystem ausgeführt werden (ist mit Tycho kompatibel)
- Kann nach kurzer Einarbeitungszeit von jedem angewendet werden
- Bietet Unterstützung bei der Migration anderer Frameworks
- Kann in Java erweitert werden

Neben den Computern mit Windows als Betriebssystem kommen auch Computer mit Linux zum Einsatz. Diese werden primär zum Bauen von MAGMASOFT® und anderen rechenintensiven Aufgaben eingesetzt. Daher gilt die Kompatibilität mit Linux als Qualitätsmerkmal. Da anzunehmen ist, dass eine kurze Laufzeit, sowie die Möglichkeit das Tool über das bestehende Buildsystem auszuführen, die Entwicklung beschleunigt, gilt dies ebenfalls als Qualitätsmerkmal. Selbes gilt für eine kurze Einarbeitungszeit. Zudem werden auch weitere Frameworks wie beispielsweise Mockito [Kac19] oder Hamcrest [LCOR20] verwendet. Es ist wünschenswert, dass diese ebenfalls mit dem verwendeten Tool migriert werden können. Das letzte Qualitätsmerkmal ergibt sich daraus, dass im Modelteam hauptsächlich Java verwendet wird. Daher ist es ebenfalls erwünscht, dass eventuell anfallende Anpassungen in Java vorgenommen werden können.

3.3 Marktanalyse

Im folgenden Abschnitt werden Werkzeuge vorgestellt, die für den Anwendungsfall der Migration von JUnit 4 nach JUnit 5 infrage kommen. Diese werden anhand der zuvor festgelegten Kriterien bewertet. Anschließend wird eines der Werkzeuge zur weiteren Evaluation ausgewählt. Grundsätzlich gibt es auch die Option, eigene Skripte zur Migration von JUnit zu schreiben. Diese Option kann zwar am besten an die individuellen Bedürfnisse anpasst werden, wäre jedoch auch mit dem größten Aufwand und dem höchsten Fehlerpotential verbunden. Da es bereits existierende Lösungen gibt, wurde die Option verworfen und wird im Folgenden nicht weiter behandelt.

Die IntelliJ IDEA [Sco20] ist eine Entwicklungsumgebung für Java und bietet auch Unterstützung bei der Migration von JUnit 4 nach JUnit 5. So kann im Kontextmenü der IDE innerhalb einer JUnit

4 Klasse die Option 'Migrate to JUnit 5' angewählt werden. Dadurch werden die Annotationen und die dazugehörigen Imports von JUnit 4 nach JUnit 5 migriert. Jedoch müssen Änderungen, wie die Position der Message oder der Wechsel von der JUnit 4 Exception-Rule zu *assertThrows(...)* immer noch vom Entwickler selbst durchgeführt werden.

Error Prone [Goo23] ist ein von Google entwickeltes Framework zur statischen Codeanalyse. Es kann über Maven, Gradle oder Ant eingebunden werden und integriert sich in den Java-Compiler, um erweiterte Fehlermeldungen auszugeben. Error Prone bietet mit dem Refaster Templates [OSG22] auch ein Feature, welches es ermöglicht, Java Sourcecode nach einem festen Muster automatisiert zu modifizieren. Innerhalb eines Templates kann mit der *@BeforeTemplate* Annotation eine oder mehrere Strukturen vorgegeben werden, welche in die mit der *@AfterTemplate* Annotation vorgegebene Struktur umgewandelt wird. Ein Beispiel dafür ist in der nachfolgenden Abbildung zu sehen. Jedoch können die Refaster Templates keine Annotation, Imports, oder Rückgabetypen ändern.

```
public class StringIsEmpty {
    @BeforeTemplate
    boolean oldStringCheck(String string) {
        return string.length() == 0;
    }

    @AfterTemplate
    boolean newStringCheck(String string) {
        return string.isEmpty();
    }
}
```

Abbildung 3.1: Beispiel eines Refaster Templates

OpenRewrite [Ope23b] ist ein Open Source Tool, welches entwickelt wurde, um aufwendige Code-Transformationen in Java-Anwendungen zu erleichtern. Dies geschieht mithilfe sogenannter Rezepte. In diesem werden die automatisierten Sourcecode Änderungen definiert. Da bereits eine große Auswahl an Rezepten besteht, könnte OpenRewrite auch für die Migration anderer Frameworks oder den Wechsel auf eine neuere Java Version nützlich sein. Eine ausführlichere Beschreibung von OpenRewrite ist im Kapitel 2.2 zu finden.

3.3.1 Auswahl eines Tools zur Evaluation

Grundsätzlich erfüllen alle drei zur Verfügung stehenden Optionen die in 3.2 festgelegten Rahmenbedingungen. Die IntelliJ IDEA bietet vor allem Unterstützung bei der manuellen Migration von einzelnen JUnit Klassen, Aufgrund der fehlenden Möglichkeit, die Änderung auf alle Klassen anzuwenden und da immer noch der Großteil der Änderungen vom Entwickler selbst ausgeführt werden muss, stellt IntelliJ keine geeignete Option zur Automatisierung der Migration von JUnit 4 nach JUnit 5 dar. Zudem fallen Lizenzgebühren von 600€ pro Entwickler jährlich an, wohingegen die Alternativen kostenlos sind. Error Prone bietet mit den Refaster Templates zwar eine Option Javacode automatisiert zu modifizieren, jedoch sind diese im Vergleich zu OpenRewrite recht begrenzt. So kann Error Prone zum Beispiel keine Imports anpassen. Zudem gibt es keine Sammlung von Refaster Templates zu Migration von JUnit oder anderen Frameworks. Daher müssten die Refaster Templates selbst geschrieben werden. OpenRewrite kann im Gegensatz zu Error Prone auf alle Teile des Javacodes zugreifen und diese modifizieren. Ein weiterer Vorteil von OpenRewrite gegenüber von Error Prone ist die Möglichkeit Tests für die Rezepte schreiben zu können. Außerdem bietet OpenRewrite auch Rezepte für andere Anwendungsfälle. Dies würde auch eine eventuell höhere Einarbeitungszeit rechtfertigen. Aufgrund der zuvor genannten Gründe wurde OpenRewrite für die weitere Evaluation ausgewählt.

4 Evaluation von OpenRewrite

In diesem Kapitel wird OpenRewrite auf seine Anwendbarkeit für die Entwicklung von MAGMASOFT® untersucht. Zu diesem Zweck sollen die bestehenden JUnit 4-Teste nach JUnit 5 migriert werden. Ziel ist es jedoch nicht, die Tests zu migrieren, sondern OpenRewrite auf seine Tauglichkeit zu untersuchen. Die Migration der Test ist daher lediglich als Nebenprodukt anzusehen.

4.1 Erste Anwendung von OpenRewrite

Um einen ersten Eindruck von OpenRewrite zu erhalten wurde, der Quickstart-Guide [Ope23e] von OpenRewrite bearbeitet. Dazu wurde das in der Dokumentation von OpenRewrite angegebene Beispiel-Projekt verwendet. Das im Quickstart-Guide angegebene Rezept zum Neuanordnen von Imports konnte ohne Probleme in die pom.xml des Projektes eingebunden werden und mit dem anschließend angegeben Befehl ausgeführt werden. Ein Vergleich der Zustände vor nach dem Anwenden von OpenRewrite zeigte, dass alle Imports wie erwartet neu angeordnet wurden.

```
mvn rewrite:run
```

Danach wurde der Guide [Ope23d] zur Migration von JUnit zu bearbeitet. Dazu wurde ein neues Maven-Projekt erzeugt und in diesem mehrere JUnit 4 Testklassen angelegt. Die in der Dokumentation angegebene Konfiguration konnte ohne Probleme zur pom.xml des Projektes hinzugefügt werden und ist in der nachfolgenden Abbildung 4.1 zu sehen. Das Ausführen des Rezeptes verlief fehlerfrei und dauerte circa 5 Sekunden. Die Inspektion der Testklassen und der pom.xml ergab, dass diese ordnungsgemäß auf die neuste JUnit 5 Version migriert wurden.

```
<plugin>
  <groupId>org.openrewrite.maven</groupId>
  <artifactId>rewrite-maven-plugin</artifactId>
  <version>5.13.0</version>
  <configuration>
    <activeRecipes>
      <recipe>org.openrewrite.java.testing.junit5.JUnit5BestPractices</recipe>
    </activeRecipes>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.openrewrite.recipe</groupId>
      <artifactId>rewrite-testing-frameworks</artifactId>
      <version>2.1.0</version>
    </dependency>
  </dependencies>
</plugin>
```

Abbildung 4.1: Code zum Einbinden von OpenRewrite in die pom.xml eines Maven-Projektes

Anschließend wurde der Guide [Ope23a] zum Schreiben und Testen eigener Rezept bearbeitet. Dabei kam es zu dem Problem, dass die Eclipse IDE manchmal die von OpenRewrite stammenden Imports nicht erkannte. Dies ist aber nicht auf OpenRewrite zurückzuführen, sondern ein Problem von Eclipse und Maven und lässt sich mit dem nachfolgenden Befehl beheben:

```
mvn eclipse:clean eclipse:eclipse
```

Das weitere Bearbeiten des Guides verlief problemlos und im Zuge dessen wurden die Beispiele in Kapitel 2.2.2 erzeugt. Zudem wurde ein Rezept zum Ausgeben des LSTs geschrieben. Dafür wurde die Anleitung zum Tree Visitig Printer [Ope23f] verwendet.

4.1.1 Erste Produktiv-Tests

Da die MAGMASOFT®-Testbasis sehr groß ist, wurde OpenRewrite zunächst an drei kleineren Projekten getestet. Diese sind eigenständige Maven-Projekte und hab mit 3 bis 15 Testklassen, eine sehr kleine Testbasis. Dies bietet den Vorteil, die migrierten Tests schnell selbst überprüfen zu können. Bei jedem der Projekte konnten die von OpenRewrite benötigten Abhängigkeiten ohne Probleme eingebunden werden. Das Ausführen von OpenRewrite verlief in allen Fällen ohne Fehlermeldung und dauerte jeweils zwischen 8 und 56 Sekunden. Bei der Inspektion des Vergleiches vor und nach dem Anwenden von OpenRewrite, ist jedoch bei zwei der Projekte ein Problem aufgefallen. Dies folgt daraus, dass das in JUnit 4 existierende Rule Feature in JUnit 5 abgeschafft wurde [Gar17]. In JUnit 4 kann beispielsweise mithilfe einer Rule überprüft werden, ob eine Exception geworfen wurde. Ein Beispiel dafür zeigt die Abbildung 4.2. In JUnit 5 verwendet dafür die auf Abbildung 4.3 gezeigte *assertThrows(...)* Methode.

```
@Rule
public ExpectedException exceptionRule = ExpectedException.none();

@Test
public void whenExceptionThrown_thenRuleIsApplied() {
    exceptionRule.expect(NumberFormatException.class);
    exceptionRule.expectMessage("For input string");
    Integer.parseInt("Not a Number.");
}
```

Abbildung 4.2: JUnit 4 Exception-Rule

```
@Test
public void whenDerivedExceptionThrown_thenAssertionSucceeds() {
    Exception e = assertThrows(RuntimeException.class, () -> {
      Integer.parseInt("Not a Number.");
    });
    assertTrue(e.getMessage().contains("For input string"));
}
```

Abbildung 4.3: JUnit 5 AssertThrows

Dies hat zur Folge, dass Testklassen mit Rules, nicht durch ein einfaches Schema von JUnit 4 nach JUnit 5 übersetzt werden können. So heißt es auch in der Dokumentation [Ope23d] unter dem Punkt Know Limitations: *Not every JUnit 4 feature or library has a direct JUnit 5 equivalent. In these cases, manual changes will be required after the automation has run.*

4.2 Anwendung auf die MAGMASOFT®-Testbasis

In diesem Abschnitt wird die Anwendung von OpenRewrite auf die MAGMASOFT®-Testbasis beschrieben. Diese verlief in mehreren Schritten. In einem ersten Testlauf wurde OpenRewrite auf die MAGMASOFT®-Tests innerhalb eines temporären Maven-Projektes angewendet. Danach wurde ein Testverfahren entwickelt, um die erfolgreiche Migration der Tests sicherstellen zu können. Anschließend wurde ein Rezept geschrieben, um die von der MAGMA Gießereitechnologie GmbH selbst entwickelten Rules zu migrieren.

4.2.1 Erster Testlauf mit MAGMASOFT®-Tests

Der MAGMASOFT® Java-Code befindet sich in Eclipse-Projekten. Diese werden mit Tycho [MFS17], einer Erweiterung für Maven, gebaut. Da zum Bauen von MAGMASOFT® einige Abhängigkeiten gegeben sein müssen, kann OpenRewrite nicht ohne weiteres mit *mvn rewrite:run* auf die Tests von MAGMASOFT® angewendet werden. Nach dem Einbinden der benötigen dependencies wurde OpenRewrite auf die ungefähr 2.000 Testklassen angewendet. Von diesen enthielten circa 800 Klassen JUnit 4 Tests. Das Ausführen von OpenRewrite hat circa 90 Minuten gedauert und verlief ohne Fehlermeldung. Ein erstes Durchschauen der Ergebnisse zeigte, dass neben den erwarteten Problematiken mit den JUnit 4 Rules, auch weiteren Probleme aufgetreten sind. Neben den Standard JUnit-Rules, werden auch selbst entwickelte Rules verwendet. Diese werden fortan als MAGMA-Rules bezeichnet und sind bereits mit JUnit 5 kompatibel. Insgesamt enthalten circa 450 der rund 800 JUnit 4 Tests eine solche MAGMA-Rule. Bei der Migration dieser Rules muss lediglich die *@Rule* Annotation zu *@RegisterExtension* geändert, sowie der dazugehörige Import angepasst werden. Dies ist jedoch nicht geschehen und scheint eine Limitierung des *JUnit5BestPractices*-Rezeptes zu sein. Des Weiteren wurden insgesamt rund 240 Klassen nicht vollständig migriert und wiesen Kompilierungsfehler auf. Bei diesen Klassen sind zum Teil die Imports nicht angepasst oder die Position der Fehlermeldung nicht geändert worden. Von diesen Klassen hat die deutliche Mehrheit eine MAGMA-Rule beinhaltet oder von einer Klasse mit solch einer Rule geerbt. Dies wirft die Frage auf, ob die fehlerhaft migrierten Klassen auf die selbstgeschriebenen MAGMA-Rules zurückzuführen sind. Alternativ könnten auch die große Anzahl an Testklassen oder das Einbinden von Eclipse Plugin Projekten als Abhängigkeiten in Maven eine Ursache sein. Aufgrund der großen Anzahl an geänderten Klassen war es nur möglich, eine stichprobenartige Überprüfung in angemessener Zeit durchzuführen. Daher ist es notwendig, für die weitere Entwicklung ein Testverfahren zu entwickeln, welches die migrierten Tests auf ihre Funktionsfähigkeit überprüft. Anschließend wurden die Änderungen an den Klassen mit einer MAGMA-Rule rückgängig gemacht. Die verbleibenden 23 Klassen mit Fehlern wurden von Hand überarbeitet.

Abschließend lässt sich nach dem ersten Anwenden von OpenRewrite auf die MAGMASOFT® Testbasis die folgende Zwischenbilanz ziehen. Das Ausführen mit Maven auf die Testbasis, hat mit 90 Minuten, eine deutlich erhöhte Laufzeit. Zudem gab es nicht vollständig migrierte Testklassen, ohne dass eine Fehlermeldung ausgegeben wurde. Daher muss ein System zum Überprüfen der migrierten Testklassen entwickelt werden. Des Weiteren sind die selbstgeschriebenen MAGMA-Rules problematisch, da diese nicht vom Standardrezept migriert werden können. Das Vorgehen bei der Migration der MAGMA-Rules ist jedoch einfach. Daher wurde beschlossen, ein eigenes Rezept zum Migrieren der Rules zu schreiben. Dies ermöglicht zudem das Evaluieren der Möglichkeit eigene Rezepte zu schreiben. Somit ergeben sich für das weitere Vorgehen die folgenden Aufgaben:

- Erstellen eines Testverfahrens zum Überprüfen der Migration der Tests
- Schreiben eines Rezeptes zum Migrieren der MAGMA-Rules
- Ausführen von OpenRewrite durch Tycho

4.2.2 Erstellen eines Testverfahrens für die Migration von JUnit

Aufgrund der im vorherigen Abschnitt beschriebenen Probleme wurde entschieden, ein Testverfahren zum Überprüfen der migrierten Tests zu entwickeln. Hierbei stößt jedoch der in Kapitel 2.1 beschriebene Refaktorisierungs-Algorithmus von Michael C. Feather an seine Grenzen. Da der zu testende Sourcecode die Tests selbst sind, ist es nicht ohne Weiteres möglich, kleinere Einheiten herauszubrechen und für diese Tests zu schreiben. Zudem wäre dies, aufgrund der Menge an Tests, mit einem enormen Aufwand verbunden.

Als grundlegende Indikatoren für die fehlerhafte Migration eines Tests können Kompilierungsfehler oder das Fehlschlagen des Tests beim Ausführen dienen. Dies ist jedoch nur eine recht grobe Überprüfung des Tests auf seine Funktionsfähigkeit. Daher wurde ein bereits existierendes Tool zum Finden von nicht ausgeführten Tests erweitert. Dieses benutzt die Codecoverage um nicht ausgeführte JUnit-Tests zu finden. Dazu wird nach dem Ausführen der JUnit Tests, die Testabdeckung aus den Ergebnissen der JUnit Tests erzeugt. Anschließend kann die Klasse zum Auswerten der Testabdeckung gestartet werden. Diese gibt eine Fehlermeldung aus, wenn weniger als 20% der Instruktionen in einer Testklasse durchlaufen werden. Die grundlegende Hypothese ist, dass eine Änderung in den durchlaufenden Instruktionen, eine fehlerhafte Migration anzeigen kann. Dazu werden vor der Migration eine Kopie der Testabdeckung angelegt. Nach der Migration werden die Tests erneut ausgeführt und eine neue Testabdeckung erzeugt. Anschließend kann die Anzahl der durchlaufenden Instruktionen verglichen werden. Grundsätzlich kann sich die Anzahl der durchlaufenden Instruktionen jedoch auch durch das Umstellen von JUnit 4 nach JUnit 5 ändern. Diese kann beispielsweise durch die Umstellung von *assertTrue(referancObject.equals(testObject))* zu *assertEquals(referancObject, testObject)* kommen. Daher ist es notwendig, für den Test eine gewisse Abweichung in der Anzahl an Instruktionen zuzulassen. Dies ist auf verschiedene Arten möglich. Zum einen durch das Betrachten der absoluten Anzahl an Abweichung und dem Festlegen einer Fehlerschranke. Zum anderen durch das Festlegen eines zulässigen Deltas, innerhalb dessen sich die relative Abweichung bewegen darf. Die absolute Betrachtung bietet den Vorteil, dass der Test beim Überschreiten einer gewissen Abweichung immer als fehlerhaft markiert wird. Da jedoch die Länge der Testklassen häufig stark voneinander abweicht, für dies falsch positiven Meldungen. Dieses Problem wird durch Betrachten der relativen Abweichung behoben. Dafür hat die relative Betrachtung grundsätzlich die Gefahr, bei sehr großen Testklassen eine Änderung nicht zu erkennen. Aufgrund der stark abweichenden Größe der Testklassen wurde sich für die Methodik, der Betrachtung der relativen Abweichung entschieden. Das Testen der Erweiterung mit den in Abschnitt 4.2.1 erstellten Daten, ergab die folgenden Ergebnisse:

- Ordnungsgemäß migrierte Klassen können eine relative Abweichung von circa 1% bis 2% in der Testabdeckung haben.
- Die Umstellung von der Exception-Rule zu *assertThrows(...)* verursacht eine relative Änderung zwischen 5% und 8%.
- Des Weiteren wurden verschiedene Fehlerszenarien getestet. Diese ergaben eine Änderung in der Testabdeckung zwischen 15% und 40%.

Da Tests mit automatisch migrierten Rules, aufgrund der zuvor beschriebenen Problematiken, noch einmal manuell überprüft werden sollen, wurde für die zulässige Abweichung ein Delta von 2% festgelegt. Abschließend wurde das entwickelte Testverfahren auf die rund 250 mit OpenRewrite nach JUnit 5 migrierten Testklassen angewendet. Dabei wurden weitere 8 Testklassen mit Fehlern gefunden. Nach dem Beheben der Fehler wurden diese für die Entwicklung von MAGMASOFT® bereit gestellt. Bei den nach dieser Methode überprüften Tests gab es bis zum Abschluss dieser Arbeit keinerlei Probleme.

4.2.3 Entwicklung eines Rezeptes zur Konvertierung von MAGMA-Rules

Anschließend wurde mit der Entwicklung eines Rezeptes zur Migration der MAGMA-Rules begonnen. Dieses soll die *@Rule* sowie *@ClassRule* Annotation zu *@RegisterExtension* ändern und die dazugehörigen Imports anpassen, wenn die Klasse keine Standard JUnit 4 Rule enthält.

Die Dokumentation von OpenRewrite [Ope23b] enthält keine vollständige Auflistung von Methoden zum Modifizieren des LST, zusammen mit einer Erklärung der Funktionalität. Daher hat es sich für die Entwicklung des Rezeptes am Hilfreichsten herausgestellt, innerhalb des GitHub Repositories [Ope23g] von OpenRewrite zum Migrieren von Test-Frameworks, nach einer Klasse mit ähnlicher Funktionalität zu suchen. Eine Suche nach 'annotation' hat zu der Klasse UpdateTestAnnotation geführt. Diese verwendet die ChangeType-Klasse, um den Typ der Annotation zu ändern. Anschließend wurde innerhalb des in 4.2.1 angelegten Maven-Projektes die Klasse *ConvertMAGMARules* erstellt. Diese ist auch in Abbildung 2.2 zu sehen. Innerhalb der Klasse wird die *visitCompilationUnit(...)* Methode aufgerufen, um zu überprüfen, ob die Klasse eine MAGMA-Rule, aber keine JUnit 4 Rules verwendet. Ist das der Fall wird in der *visitAnnotation(...)* Methode die *@Rule* beziehungsweise *@ClassRule* Annotation zu *@RegisterExtension* geändert. Dies geschieht durch das Erzeugen eines neuen Visitors mithilfe der ChangeType-Klasse. Der erzeugte Visitor wird mit der *doAfter(...)* Methode dem Rezept hinzugefügt. Die sieht beispielsweise für die *@Rule* Annotation wie folgt aus:

```
doAfterVisit(new ChangeType(JUNIT_RULE, JUNIT_EXTENSION, true).getVisitor());
```

Zudem wurde eine Klasse *ConvertMAGMARulesTest* angelegt. Diese ist auszugsweise in 2.3 zu sehen. Die Klasse testet sowohl die Szenarien in dem die *@Rule* beziehungsweise *@ClassRule* Annotation zu *@RegisterExtension* geändert werden sollen, als auch die Szenarien, in denen keine Änderungen vorgenommen werden sollen. Für die Tests, in denen die Annotationen geändert werden sollen, wurde für jede MAGMA-Rule ein Test angelegt. Des Weiteren wurden auch drei Testfälle mit Kombinationen verschiedener MAGMA-Rules erstellt. Für die Fälle, in denen die Annotation nicht geändert werden soll, wurde ein Testfall für jede Standard JUnit 4 Rule erstellt. Außerdem wurden drei Testfälle mit Kombinationen von JUnit 4 Rules, sowie drei Testfälle mit Kombinationen von JUnit 4 Rules und MAGMA-Rules erstellt.

4.2.4 Anwendung von OpenRewrite mit Tycho

Bei der Entwicklung von MAGMASOFT® wird das Eclipse Framework [TH12] verwendet. Daher befindet sich der Javacode in mehreren Eclipse-Plugin-Projekten. Diese werden mit Tycho [MFS17] gebaut. In diesen Abschnitt wird das Ausführen des zuvor erstellten OpenRewrite Rezeptes mit Tycho erläutert.

Tycho erweitert Maven mit mehreren Plugins um Eclipse Projekte zu bauen. Daher fällt der Javacode von MAGMASOFT® in die Kategorie der Multi-Module Maven Projekte. Für diese wird in der Dokumentation von OpenRewrite [Ope23i] empfohlen, in der parent pom.xml ein Profile anzulegen. Anschließend soll es möglich sein OpenRewrite aus jedem Untermodule mit dem folgendem Befehl auszuführen.

```
mvn -Popenrewrite rewrite:run
```

Da auf den Entwicklungsrechnern jedoch aus Sicherheitsgründen das Internet abgeschaltet ist, kann der Befehl nicht ohne weiteres verwendet werden. Deshalb wurde der Befehl nicht direkt ausgeführt. Stattdessen wurde eine bestehendes Buildskript erweitert. Dieses konfiguriert den Build so, dass der Interne Nexus-Server verwendet wird. Durch dieses Vorgehen wird jedoch OpenRewrite bei jedem Build ausgeführt. Da die Änderungen jedoch nur lokal sind und durch das Versionsverwaltungs-System rückgängig gemacht werden können, wird dies für die Evaluation von OpenRewrite als akzeptabel angesehen. Eine langfristige Lösung wäre, zum Beispiel der Einbau eines Mechanismus, welches OpenRewrite nur bei der Übergabe eines Parameters ausführt. Beim Ausführen des Builds war zu erkennen, dass OpenRewrite gestartet wird und die benötigten Abhängigkeit von Nexus-Server heruntergeladen wurden. Jedoch wurde das im Profile definiert Rezept nicht ausgeführt. Stattdessen konnte mit Hilfe eines anderen Vorgehens das Rezept ausgeführt werden. Dabei wird kein Profile angelegt, sondern OpenRewrite direkt in die Aggregator pom.xml von MAGMASOFT® hinzugefügt. Anstatt das Rezept direkt zu aktiveren, wird dieses beim Aufruf von Maven mit dem folgenden Befehl getan.

```
mvn rewrite:runNoFork -Drewrite.activeRecipes=$RECIPE_NAME
```

Hierbei wurde der Aufruf *rewrite:runNoFork* verwendet, um potenzielle Probleme durch Forking im Maven-Lifecycle zuvor zukommen. Zudem steht *$RECIPE_NAME* hierbei als Platzhalter für den full qualified class name des ausgeführten Rezeptes. Um das in Abschnitt 4.2.3 erstellte Rezept ausführen zu können, muss dieses verfügbar gemacht werden. Dies geht über das Deployen auf einen Nexus-Server oder durch die Installation in das lokale Maven-Repository. Um unnötige Komplexität zu vermeiden, wurde sich für das lokale Installieren entschieden. Bei der Entwicklung von MAGMASOFT® hat jeder Workspace sein eigenes Maven-Repository, daher muss dieses konfiguriert werden. Da eine solche Konfiguration bereits besteht, muss diese lediglich bei der Installation mit dem folgenden Befehl mit angegeben werden.

```
mvn –settings $PATH_TO/maven-settings.xml install
```

Anschließend konnte das Rezept zum Ändern der MAGMA-Rules angewendet werden. Das Ausführen mit Tycho dauerte circa 20 Minuten. Die folgende stichprobenartige Sichtung hat ergeben, dass die *@Rule* Annotationen in allen kontrollierten Klassen ordnungsgemäß zu *@RegisterExtension* geändert wurden. Das weitere Vorgehen würde vorsehen, dass *org.openrewrite.testing.JUnit5*-Rezept auf die Tests mit den bereits migrierten MAGMA-Rules erneut anzuwenden. Anschließend würden die nach JUnit 5 migrierten Tests mit den in Abschnitt 4.2.2 beschrieben Testverfahren überprüft. Das Hauptziel der Arbeit ist jedoch OpenRewrite auf seine Wirtschaftlichkeit zu überprüfen und nicht die JUnit Tests zu migrieren. Daher wurde sich gegen die Aufnahme der weiteren Schritte in diese Arbeit entschieden.

5 Diskussion der Ergebnisse

In diesem Kapitel werden die Ergebnisse aus Kapitel 4 ausgewertet. Dazu werden die in Kapitel 3.2 festgelegten Kriterien zur Bewertung herangezogen. Anschließend werden die in Kapitel 4 festgestellten Limitierungen heraus gearbeitet, um abschließend die Wirtschaftlichkeit von OpenRewrite bewerteten zu können.

5.1 Auswertung der Ergebnisse

OpenRewrite kann durch Maven eingebunden und ausgeführt werden. Dadurch werden die Rahmenbedingungen, also die Kompatibilität mit Windows und die Funktionsfähigkeit ohne direkten Internetzugriff, erfüllt. Für den Einsatz von OpenRewrite ohne direkten Zugriff auf das Internet muss jedoch ein Nexus-Server [Var19b] verwendet werden. Dieser ist bereits bei der MAGMA Gießereitechnologie GmbH vorhanden. Zusätzlich werden durch das Verwenden von Maven die Qualitätsmerkmale der Kompatibilität mit Linux und dem bestehenden Buildsystem von MAGMASOFT® erfüllt. OpenRewrite ist grundsätzlich quelloffen und somit kostenfrei. Damit ist die Rahmenbedingung von Kosten unter ein bis zwei Monatsgehältern eines Entwicklers erfüllt. Die Laufzeit von 15 bis 20 Minuten bei der Anwendung auf die gesamte MAGMASOFT® Codebasis ist, laut den Entwicklern, für die Migration von JUnit akzeptabel. Des Weiteren bietet OpenRewrite eine Vielzahl an Rezepten, die die Entwicklung von MAGMASOFT® beschleunigen können. Darunter sowohl Rezepte für die Migration auf ein neuere Java Version, als auch für die Migration anderer Frameworks wie Mockito oder Hamcrest. Somit ist das Qualitätsmerkmal der weiteren Anwendungsbereiche ebenfalls erfüllt. Die Dokumentation von OpenRewrite unterstützt mit einer Vielzahl an Anleitungen vor allem die Installation und Inbetriebnahme der Software. Für das Schreiben eigener Rezepte in Java gibt es jedoch abgesehen von einer allgemeinen Einführung keine weiteren Hilfestellungen. Grundsätzlich sind jedoch die Qualitätsmerkmale, sowohl für die Einarbeitungszeit als auch für die Erweiterbarkeit in Java erfüllt. Die nachfolgende Abbildung 5.1 zeigt die für die einzelnen Arbeitsschritte benötigte Zeit, wobei eine Zelle in der Tabelle einen Arbeitstag entspricht. Für die verbleibenden Arbeitsschritte (in grau) wird die benötigte Zeit auf zwei Tage geschätzt. Somit ist der gesetzte zeitliche Rahmen, von ein bis zwei Monaten, erfüllt.

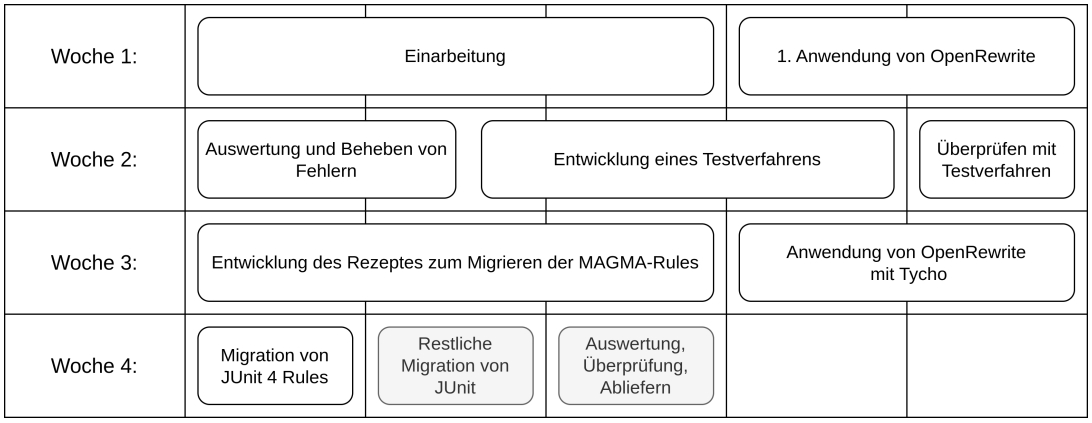

Abbildung 5.1: Darstellung des Zeitaufwands zur Migration der JUnit Tests

5.2 Limitierungen von OpenRewrite

Bei der Anwendung von OpenRewrite hat sich als größte Limitierung herausgestellt, dass OpenRewrite lediglich eine Ausgabe darüber gibt, welche Klassen geändert wurden. So wird keine Fehlermeldung ausgegeben, wenn Klassen nur unvollständig migriert werden. Dies ist bei kleinen Codemengen unproblematisch, da diese schnell kontrolliert werden können. Bei größeren Codemengen ist dies jedoch unpraktikabel. Daher ist es wichtig, die Funktionsfähigkeit durch zum Beispiel Unit-Tests sicherzustellen. Eine weitere Limitierung von OpenRewrite besteht darin, dass es nur auf ganze Projekte angewendet werden kann. So ist es nicht möglich OpenRewrite nur auf einzelne Klassen anzuwenden. Zudem bleiben noch Limitierungen, welche technologieunabhängig sind und daher kommen, dass bestimmte Prozesse nur schwer generalisiert und somit auch nicht gut automatisiert werden können. Ein Beispiel dafür sind die JUnit 4 Exception-Rules. Zwar ist es möglich, diese mit OpenRewrite nach JUnit 5 zu migrieren, dies jedoch nur rudimentär. Der Grund dafür ist jedoch keine technologische Limitierung von OpenRewrite, sondern dass nur schwer ein generelles Schema für die Automatisierung abgeleitet werden kann.

5.3 Bewertung der Wirtschaftlichkeit von OpenRewrite

Die Auswertung in Abschnitt 5.1 hat gezeigt, dass OpenRewrite sowohl die Rahmenbedingungen als auch die Qualitätsmerkmale der MAGMA Gießereitechnologie GmbH erfüllt. Zudem ist davon auszugehen, dass es möglich ist, mit OpenRewrite die benötigte Zeit zur Migration der MAGMASOFT® Testbasis von JUnit 4 nach JUnit 5, bedeutend zu verkürzen. OpenRewrite hat zwar Limitierungen, wie eine fehlende Validierung des migrierten Sourcecodes. Diese lässt sich jedoch durch die bestehenden Unit-Tests lösen und im Fall der Migration von JUnit, kann sie durch das in Abschnitt 4.2.2 beschriebene Tool behoben werden. Abschließend lässt sich sagen, dass OpenRewrite ein großes Potenzial hat, um repetitive Prozesse zu automatisieren. Dies funktioniert umso besser, je eindeutiger die Regeln sind, nach denen gehandelt wird. Jedoch müssen gegebenenfalls Maßnahmen ergriffen werden, um die Funktionsfähigkeit des Sourcecodes sicherzustellen. Somit ist davon auszugehen, dass OpenRewrite einen wirtschaftlichen Mehrwert für die Entwicklung von MAGMASOFT® bieten kann.

6 Fazit und Ausblick

In diesem Kapitel wird ein Fazit aus dem vorherigen Kapitel abgeleitet. Abschließend wird ein Ausblick auf die weiteren Anwendungsmöglichkeiten von OpenRewrite gegeben.

6.1 Fazit

OpenRewrite bietet eine Vielzahl an Funktionen zur Modifikation großer Mengen an Sourcecode. Dies funktioniert umso besser, je einfacher die Regeln sind, nach denen die Änderung durchgeführt werden. Dabei ist jedoch zu beachten, dass OpenRewrite lediglich ausgibt, an welchen Klassen Änderung durchgeführt wurden. Daher muss die Funktionsfähigkeit der Klassen, beispielsweise durch Unit-Tests, sichergestellt werden. Des Weiteren gibt es bereits eine Vielzahl an Rezepten zur Migration von Frameworks, der Migration zu einer neueren Java Version und weiterer Anwendungsfälle. Abschließend lässt sich sagen OpenRewrite hat ein großes Potenzial, um einfache, aber häufig anfallende Sourcecode Änderungen zu automatisieren. Dies ermöglicht es dem Entwickler, Zeit und somit Kosten zu sparen.

6.2 Ausblick

Für die weitere Anwendung von OpenRewrite muss das Buildsystem von MAGMASOFT® zum Ausführen von OpenRewrite angepasst werden. Dabei ist es wünschenswert, OpenRewrite gezielt auf einzelne Projekte und nicht nur auf den gesamten MAGMASOFT® Java-Sourcecode anwenden zu können. Diese Option alleine kann bereits das Ausführen von OpenRewrite signifikant beschleunigen. Des Weiteren müsste eine Dokumentation für das Anwenden und Entwickeln von OpenRewrite im MAGMASOFT®-Wiki angelegt werden. Nach Umsetzung dieser Maßnahmen ergeben sich drei Anwendungsbereiche für OpenRewrite. Zum einen das Anwenden der Rezepte zur Aktualisierung anderer Frameworks oder zum Wechsel auf die neuste Java Version. Da Java rückwärts kompatibel ist, führt das reine Hochzählen der Versionsnummer nicht unbedingt zu etwaigen Performanceverbesserung. Somit kann die Aktualisierung des Bestandscodes sowohl eine Verbesserung für die Entwicklung als auch für die Performanc von MAGMASOFT® bedeuten. Darauf aufbauend kann auch eine CI-Pipeline entwickelt werden, welche sicherstellt, das neu geschriebener Sourcecode immer den aktuellen Standard entspricht. Zudem können auch eigene Rezepte geschrieben werden, um häufig anfallende Aufgaben zu vereinfachen. Ein Beispiel hierfür ist eine häufig verwendete Klasse, bei der ein Setter durch ein Argument in Konstruktor ersetzt werden soll. Diese Art an Änderungen fallen häufig bei Arbeiten am Codegenerator an. Um die Anwendung dieser Rezepte für den Entwickler zu erleichtern, kann ein Eclipse Plugin geschrieben werden, welches es ermöglicht OpenRewrite aus der Eclipse IDE heraus aufzurufen.

A Literaturverzeichnis

[Bec02] Kent Beck. *Test driven development*. The Addison-Wesley signature series. Addison-Wesley Educational, Boston, MA, November 2002.

[Che17] Fu Cheng. *The Platform Logging API and Service*. Apress, New York, 1 edition, December 2017.

[ES13] Karl Eilebrecht and Gernot Starke. *Patterns Kompakt*. It Kompakt. Springer Vieweg, Berlin, Germany, 4 edition, May 2013.

[Fea04] Michael C. Feathers. *Working Effectively with Legacy Code*. Robert C. Martin series. Prentice Hall, Upper Sadle River, NJ 07458, October 2004.

[Fou] Eclipse Foundation. Eclipse IDE. eclipse.org/ide.

[Fow19] Marthin Fowler. *Refactoring: Improving the Design of Existing Codeoring*. Addison Wesley, 2 edition, January 2019.

[Gar17] Boni Garcia. *Mastering Software Testing with JUnit 5 - Comprehensive guide to develop high quality Java applications*. Packt Publishing Ltd, Birmingham, October 2017.

[Gmb] MAGMA Gießereitechnologie GmbH. Magmasoft®. magmasoft.de.

[Goo23] Goolge. Error prone. errorprone.info, 2023.

[JUn23a] JUnit.org. Junit 4. junit.org/junit4, December 2023.

[JUn23b] JUnit.org. Junit 5. junit.org/junit5, December 2023.

[Kac19] Tomek Kaczanowski. *Practical unit testing with JUnit and Mockito*. Kaczanowski, Tomek, November 2019.

[LCOR20] Maurizio Leotta, Maura Cerioli, Dario Olianas, and Filippo Ricca. Two experiments for evaluating the impact of hamcrest and AssertJ on assertion development. *Software Quality Journal*, 28(3):1113–1145, September 2020.

[MFS17] Sascha Müller, Philipp M. Fischer, and Tobias Schlauch. About tycho, maven, p2 and target-platforms: From pain to best practice. In *EclipseCon*, October 2017.

[Mod] Moderne. Automated code remediation. moderne.io.

[Mus14] Benjamin Muschko. *Gradle in Action*. Simon and Schuster, New York, March 2014.

[Noo85] Robert E. Noonan. An algorithm for generating abstract syntax trees. *NASA Langley Research Center: Computer Languages*, 10:225–236, 1985.

[Ope23a] OpenRewrite. Authoring Recipes. docs.openrewrite.org/authoring-recipes, November 2023.

[Ope23b] OpenRewrite. Introduction to openrewrite. docs.openrewrite.org, November 2023.

[Ope23c] OpenRewrite. Lossless Semantic Trees (LST). docs.openrewrite.org/concepts-explanations/lossless-semantic-trees, November 2023.

[Ope23d] OpenRewrite. Migrate to JUnit 5 from JUnit 4. docs.openrewrite.org/running-recipes/popular-recipe-guides/migrate-from-junit-4-to-junit-5, November 2023.

[Ope23e] OpenRewrite. Quickstart: Setting up your project and running recipes. docs.openrewrite.org/running-recipes/getting-started, November 2023.

[Ope23f] OpenRewrite. Recipe VisitingPrinter. docs.openrewrite.org/concepts-explanations/tree-visiting-printer, November 2023.

[Ope23g] OpenRewrite. rewrite-testing-frameworks. github.com/openrewrite/rewrite-testing-frameworks, November 2023.

[Ope23h] OpenRewrite. Running Recipes. docs.openrewrite.org/running-recipes, November 2023.

[Ope23i] OpenRewrite. Running Rewrite on a multi-module Maven project. docs.openrewrite.org/running-recipes/multi-module-maven, November 2023.

[OSG22] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. Towards automated library migrations with error prone and refaster. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1598–1606, New York, USA, 2022. Association for Computing Machinery.

[SBPG18] Alex Soto Bueno, Jason Porter, and Andy Gumbrecht. *Testing Java Microservices: Using Arquillian, Hoverfly, AssertJ, JUnit, Selenium, and Mockito*. Manning Publications, New York, October 2018.

[Sco20] Helen Scott. The IntelliJ IDEA Blog: Migrating from JUnit 4 to JUnit 5. blog.jetbrains.com/idea/2020/08/migrating-from-junit-4-to-junit-5, August 2020.

[TH12] Marc Teufel and Jonas Helming. *Eclipse 4 - Rich Clients mit dem Eclipse 4.2 SDK*. Entwickler.press, Unterhaching, 2012.

[Var19a] Balaji Varanasi. *Introducing Maven - A Build Tool for Today's Java Developers*. Apress, New York, 2 edition, November 2019.

[Var19b] Balaji Varanasi. *Introducing Maven - A Build Tool for Today's Java Developers*, pages 107–113. Apress, New York, 2 edition, November 2019.

[vM23] Henry van Merode. *Continuous integration (CI) and continuous delivery (CD)*. Apress, Berlin, Germany, 1 edition, March 2023.

B Abbildungsverzeichnis