

Technische Universität München

Fakultät für Informatik

Diplomarbeit

**Messung von Zuverlässigkeits- und
Sicherheitsmethoden**

Wahid Khachabi

Aufgabensteller: Prof. Dr. Dr. hc. Manfred Broy

Betreuer: Stefan Wagner, Jan Jürjens

Abgabedatum: 15. Mai 2004

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum

Unterschrift

Inhaltsverzeichnis

1. Einleitung.....	4
2. Grundlagen.....	6
2.1. Zuverlässigkeit.....	6
2.2. Sicherheit.....	8
2.3. Zuverlässigkeits- und Sicherheitsmessung: Metriken und Modelle	10
3. Methodenauswahl	12
3.1. Gerichtete Tests	13
3.1.1. Inspektionen	13
3.1.2. Black-Box Tests.....	14
3.1.3. Glas-Box Tests	15
3.2. Repräsentative Tests.....	17
3.3. Das Experiment	18
4. Fallstudie „Verkehrsleittechnik“	19
4.1. Beschreibung des technischen Systems	19
4.2. Beschreibung der Systemrealisierung	22
4.2.1. Die Simulation	22
4.2.2. Die Implementierung.....	23
5. Methodenanwendung	35
5.1. Inspektion	35
5.2. Black-Box Tests.....	42
5.3. Glas-Box Tests	50
5.4. Repräsentative Tests.....	55
6. Ergebnisanalyse	61
6.1. Zuverlässigkeitanalyse	61
Tabelle 8. : Ergebnisanalyse für die Zuverlässigkeit.....	61
Tabelle 9. :Effizienz der Methode bei der Zuverlässigkeitsmessung	63
6.2. Sicherheitanalyse	64
Tabelle 10. : Ergebnisanalyse für die Sicherheit.....	64
Tabelle 11. : Effizienz der Methode bei der Zuverlässigkeitsmessung	65
7. Abschluss	67
Bibliographie / Links	69

1. Einleitung

Qualität ist ein Maß, das bei jedem Produkt, auch bei Software, eine Hauptrolle spielt. Die Qualität eines Produktes entscheidet nicht nur über sein Durchsetzungsvermögen auf dem Markt, sondern auch über das Image des Herstellers.

Aus diesem Grund wird der Software-Qualität immer mehr Achtung gewidmet. Dementsprechend steigen die Kosten und der Zeitaufwand, die die Software-Hersteller diesem Bereich zuweisen.

Nach [8] gibt es einige Faktoren, aus denen die Software-Qualität besteht :

- **Funktionalität:** Vorhandensein von Funktionen mit festgelegten Eigenschaften. Diese Funktionen erfüllen die definierten Anforderungen.
- **Zuverlässigkeit:** Fähigkeit der Software, ihr Benutzungsprofil unter festgelegten Bedingungen in einem festgelegten Zeitraum zu halten.
- **Benutzbarkeit:** Aufwand, der zur Benutzung erforderlich ist, und individuelle Beurteilung der Benutzung durch eine festgelegte oder vorausgesetzte Gruppe.
- **Effizienz:** Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen.
- **Änderbarkeit:** Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist. Änderungen können Korrekturen, Verbesserungen oder Anpassungen an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen einschließen.
- **Übertragbarkeit:** Eignung der Software, von einer Umgebung in eine andere übertragen zu werden.

Um Software-Qualität zu beschreiben wird im Allgemeinen Zuverlässigkeit verwendet. Weil sie auf dem Auftreten der sichtbaren Probleme oder der Ausfälle des Produkts basiert, gehört sie zu dem einfachsten Faktoren, um Qualität zu messen. Es ist weiterhin ein einfaches Konzept, worauf sich ein Benutzer beziehen kann [1].

Der Benutzer erwartet ein fehlerfreies Produkt mit hohem Zuverlässigkeitsgrad. Dies macht aus der Zuverlässigkeit, auch seitens des Benutzers, einen für die Qualität stellvertretenden Faktor.

Motivation und Ziel jeder Software-Entwicklung muss es sein, ein möglichst fehlerfreies Software-Produkt zu erstellen. Doch EDV-Software erweist sich immer wieder als schwaches Glied der Datenverarbeitung. Software ist in der Regel zu teuer und mit zu vielen Fehlern behaftet. Die wirtschaftliche Bedeutung einer Verbesserung der Softwarequalität wird dadurch unterstrichen, dass ein überproportionaler Anstieg der Softwarekosten an den Gesamtkosten von technischen Systemen zu beobachten ist. Eine Verbesserung der Zuverlässigkeit von Software führt zum Einen zur Senkung der Wartungs- und Reparaturkosten und zum Anderen werden durch eine erhöhte Zuverlässigkeit Kosten verringert, die der Ausfall eines Systems mit sich bringt. Diese Kosten sind zumeist schwerwiegender als diejenigen für Wartung und Reparatur. Auf der anderen Seite erhöhen sich durch verstärkte Zuverlässigkeitsbemühungen die Kosten für die Planung und Implementierung des Systems, so dass die Gesamtkosten nur durch eine ausgewogene Kalkulation von Aufwand und Nutzen minimiert werden können [9].

Doch es gibt mehrere Methoden zur Verbesserung von Softwaresicherheit- und Zuverlässigkeit. In dieser Arbeit sollen verschiedene Methoden verglichen werden. Mit verschiedenen Metriken lassen sich die Auswirkungen dieser Methoden messen. Es sollen zuerst Methoden ausgewählt werden, die anschließend verglichen werden sollen. Die ausgewählten Methoden sollen am Beispiel der Entwicklung einer Sicherheitskritischen Anwendung aus der Verkehrsleittechnik erprobt werden und anschließend auf Wirkung und Effizienz bewertet werden.

2. Grundlagen

2.1. Zuverlässigkeit

Wer sich mit dem Studium der Sicherheit und Zuverlässigkeit beschäftigt stellt fest, dass vertraute Alltagsbegriffe mit teilweise feinsinnigen Bedeutungsunterschieden besetzt sind, so dass es hier sinnvoll erscheint, zunächst einmal einen Überblick über dieses Vokabular zu geben. Was haben Begriffe wie „Zuverlässigkeit“, „Verlässlichkeit“, „Korrektheit“, „Sicherheit“ und „Qualität“ miteinander gemeinsam und was unterscheidet diese Begriffe?

Nach [4] lassen sich die meisten dieser Begriffe unter dem Oberbegriff „Qualitätsmerkmale“ eines Produktes zusammenfassen. Die folgenden kurzen Definitionen aus dem Bereich der Zuverlässigkeit sollen dazu dienen, diesen Begriffen etwas mehr Kontur zu verleihen :

- **Ein Ausfall (Failure)** ist eine unzulässige Abweichung einer oder mehrerer Eigenschaften, welche die Unterscheidung und Beurteilung von Betrachtungseinheiten ermöglichen.
- **Ein Fehler (Fault)** ist ein durch Programmierung bzw. Modellierung oder Spezifikation eingetretener Defekt.
- **Korrektheit (Fehlerfreiheit)** ist die Übereinstimmung zwischen realisierter und spezifizierter Funktion. Ein Softwareprogramm ist korrekt, wenn es frei ist von logischen Fehlern. Korrektheit ist also stets bezogen auf eine Aussage über die Absicht oder den Zweck eines Softwareprogramms.
- **Qualität** ist die Gesamtheit aller Merkmale einer Einheit, bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen.
- **Zuverlässigkeit** ist die Fähigkeit einer Einheit, denjenigen durch den Verwendungszweck bedingten Anforderungen zu genügen, die an das Verhalten ihrer Eigenschaften während einer gegebenen Zeitdauer gestellt sind. Im Gegensatz zur Funktionsfähigkeit, welche lediglich den momentanen Zustand eines Systems betrachtet, wird bei der Zuverlässigkeit ein bestimmtes

Zeitintervall zur Beurteilung herangezogen. Der Begriff der Zuverlässigkeit hat zudem eine weitere Bedeutung. Man bezeichnet nämlich mit Zuverlässigkeit auch den Grad des Vertrauens, welches aufgrund geringer Versagenswahrscheinlichkeit oder geringer Ausfall- bzw. Versagensrate gerechtfertigt erscheint. Diese Art der Zuverlässigkeit beschreibt man auch als Verlässlichkeit. Ein Softwareprogramm ist zuverlässig, wenn es in allen Situationen, in denen es benutzt wird sinnvolles Verhalten zeigt, d.h. Zuverlässigkeit ist hier ein Maß für die Güte der Spezifikation.

2.2. Sicherheit

Die Software Sicherheit scheint, für Anfänger auf dem Gebiet der Softwaresicherheit- und Zuverlässigkeit, ein untergeordnetes Problem der Softwarezuverlässigkeit zu sein. Das stimmt auch aus der Sicht des Softwareentwicklers, denn für ihn zählen nur die auftretenden Ausfälle und nicht unbedingt ihre Folgen.

Der Benutzer sieht es meist ganz anders, denn es könnte über Menschenleben entscheiden, ob z.B. bei einer Zugdurchfahrt ein Bahnübergang geschlossen wird.

Aus diesem Grund ist es wichtig die Softwaresicherheit von der Softwarezuverlässigkeit abzukoppeln. Einerseits wird die wichtige Bedeutung der Sicherheit bei der Softwareentwicklung dadurch unterstrichen, andererseits kann gezielt gemessen werden, inwiefern eine Software Menschen- und Sachschäden verursachen kann.

Formal ist die Sicherheit die Abwesenheit von Gefahr und wird durch die Erfüllung sicherheitsbezogener Korrektheits- und Zuverlässigkeitsforderungen erreicht.

Gerade bei dieser Arbeit steht der Begriff Sicherheit im Mittelpunkt, da eine sicherheitskritische Software entwickelt werden muss.

Im weiteren Verlauf, setzt sich diese Arbeit im Wesentlichen mit der Zuverlässigkeit auseinander. Dies ist jedoch kein Widerspruch, denn die Zuverlässigkeit ist ein übergeordnetes Problem, bezogen auf die Sicherheit.

Dies bedeutet aber nicht, dass die Sicherheit total ignoriert wird, sondern nur, dass die Sicherheitsmessung erst bei der Ergebnisanalyse zur Geltung kommen wird, denn eine Aussage über die Softwarezuverlässigkeit impliziert nicht unbedingt eine wahre Aussage über die Softwaresicherheit.

Es könnte sein, dass eine Software einen höheren Maß an Zuverlässigkeit bietet, aber die wenigen Ausfälle die noch in der Anwendung auftreten, sehr Sicherheitskritisch sein könnten.

Um die Softwaresicherheit messen zu können, werden den auftretenden Ausfällen Schweregrade zugewiesen, die aussagen, wie schwerwiegend die Ausfälle sind. Es

werden Werte zwischen 0 und 10 benutzt, 0 steht für einen ausfallfreien Testverlauf und 10 für einen sehr sicherheitskritischen Ausfall.

Außerdem ist die Häufigkeit des Auftretens eines Ausfalls auch entscheidend für die Messung der Sicherheit.

Doch die Sicherheit ist ein Abstrakter Begriff. Sicherheit kann man nicht messen, dafür brauchen wir an dieser Stelle ein Maß, das stellvertretend eine Aussage über die Sicherheit liefert.

Um die Sicherheit beschreiben zu können, wird hier ein weiterer Begriff definiert:

Risiko ist eine Metrik, die aussagt, wie sicher eine Software ist und es wird.

Das Risiko R bei einem Ausfall mit dem **Schweregrad S** hängt von der **Häufigkeit H** des Auftretens dieses Ausfalles ab:

$$R = S * H$$

2.3. Zuverlässigkeits- und Sicherheitsmessung: Metriken und Modelle

In diesem Abschnitt werden Vorgehensweisen erläutert, die dazu dienen, die Sicherheit und Zuverlässigkeit zu messen.

Um eine Messung zu erreichen, müssen zuerst Methoden angewandt werden. Diese Methoden werden im nächsten Kapitel „3“ genau erklärt.

Im Weiteren werden Vorgehensweisen, die dazu dienen, Zuverlässigkeit und Sicherheit zu verbessern, Methoden genannt. Andere Vorgehensweisen, die dazu dienen Zuverlässigkeit und Sicherheit zu messen werden Modelle genannt.

Nach der Anwendung von Sicherheits- und Zuverlässigkeitsmethoden verbessern sich die Softwaresicherheit- und Zuverlässigkeit (jedoch könnten sie sich auch verschlechtern! Denn die angewandten Methoden fordern eine Code-Änderung, die wiederum von einem Entwickler durchgeführt wird.). Um diese Verbesserung in Zahlen auszudrücken und sie anschließend analysieren zu können, bieten sich mehrere Vorgehensweisen an.

Zur Zuverlässigkeitsmessung sind „Ausfallbasierte Modelle“ (Reliability Growth Models) am verbreitetsten. Die ausfallbasierten Modelle untersuchen das Auftreten von Ausfällen bei der Software und analysieren das Ausfallverhalten.

Eine möglicher Ansatz ist „time-between-failure“ (TBF); dieses Maß dokumentiert den Zeitabstand zwischen zwei Ausfällen.

TBF kann auch als statistisches Maß benutzt werden, um das Ausfallverhalten einer Software vorherzusehen. Dieser Aspekt ist aber für diese Arbeit uninteressant, da uns viel mehr interessiert, die Möglichkeit eine Aussage machen zu können über die Zuverlässigkeit einer Software, um Ergebnisse mehrerer Zuverlässigkeits- und Sicherheitsmethoden vergleichen zu können.

Der Mittelwert der TBF binnen eines bestimmten Zeitraumes ist die „Mean Time To Failure“ (MTTF).

Ebenfalls häufig benutzt ist das „Failure Intesity“ Maß. Es besagt, wieviele Ausfälle innerhalb einer bestimmter Zeitspanne auftreten.

Wir werden aber für die Zuverlässigkeitsmessung das MTTF benutzen, da es das Maß ist, das am verbreitetsten ist.

Wie vorher erwähnt, kann man die Softwaresicherheit nicht messen. Um eine Aussage über die Sicherheit einer Software zu machen, muss das Risiko dieser Software gemessen werden. Dafür gibt es mehrere Möglichkeiten. Ein verbreiteter Ansatz dafür ist die „Expert Opinion“. Wie es der Name verrät, werden hier Experten nach dem Schweregrad von Ausfällen befragt, um diese einzustufen.

Um das Risiko zu messen benutzen wir eine einfache Metrik. Diese Metrik wird einfach aus der Summe der Risiken aller Ausfälle der Software errechnet.

Das Risiko eines Ausfalls ist definiert als der Schweregrad des Ausfalls multipliziert mit der Häufigkeit des Auftretens. Da wir jeden Test nur einmal ausführen werden, wird die Häufigkeit in unserem Fall immer 1 betragen.

Die Sicherheitsmessung ist die Summe der Risiken aller Ausfälle; in unserem Experiment ist das Risiko immer gleich dem Schweregrad des Ausfalls (weil die Häufigkeit immer 1 beträgt). Deshalb wird die Sicherheitsmessung als Summe aller Schweregrade der aufgetretenen Ausfälle berechnet.

Wir möchten nicht nur Die Auswirkung von sicherheits- und Zuverlässigkeitsmethoden studieren, sondern auch die Effizienz der angewandten Methoden. Dafür benutzen wir zwei Methoden, die auf dem selben Prinzip beruhen. Es wird immer die gemessene Verbesserung, die durch die Anwendung einer Methode bewirkt wurde durch die Zeit geteilt, die die jeweilige Methode in Anspruch nahm.

3. Methodenauswahl

Es ist sehr wichtig, dass nicht nur die aufgetretenen Ausfälle protokolliert werden, sondern auch der Schweregrad der einzelnen Ausfälle. Wie vorher erwähnt, schwankt dieser Schweregrad zwischen 0 (kein Ausfall) und 10 (sehr sicherheitskritischer Ausfall).

Neben der Untersuchung der Sicherheit, soll auch die Effizienz jeder Methode untersucht werden, und dafür sollen die genauen Zeiten, die für jede Methode aufgewandt wurden aufgenommen werden. Denn auch wie gut eine Methode ist, sondern auch wie teuer sie war, hier in Form von Zeitaufwand ausgedrückt, ist relevant.

Die Methoden werden erst einmal in zwei Hauptgruppen aufgeteilt: gerichtete und repräsentative Tests.

3.1. Gerichtete Tests

Gerichtete Tests versuchen alle Möglichkeiten einer Software auszuschöpfen, sie nehmen weniger Rücksicht auf die Spezifikation, bzw. die spätere Anwendung der Software. Es werden bei gerichteten Tests auch Ausnahmefälle oder Grenzfälle miteinbezogen. Ein einfaches Beispiel dafür wäre im Fall der Verkehrsleittechnik-Software (die Referenzfallstudie wird im Kapitel 4 näher erklärt.), dass eine Weiche und ein Bahnübergang auf derselben Position liegen. Dieser Fall wird zwar in der Realität so gut wie nie auftreten, wäre jedoch ein angebrachter gerichteter Test.

Es ist selbstverständlich, dass, je umfangreicher die Methoden sind die auf die Software angewandt werden, desto sicherer und zuverlässiger die Software nach Abschluss dieser Methoden wird. Aus diesem Grund werden drei dieser Methoden in dieser Arbeit verglichen, um Ihre Effektivität und Nutzbarkeit im Bezug auf diese Referenzfallstudie festzustellen.

3.1.1. Inspektionen

Die Inspektion ist eine übliche Methode für das Ermitteln und das Beheben von Defekten in der Softwareentwicklung, sowie beim Verhindern, dass Softwaredefekt bei der Entwicklungsphase durchgereicht werden. Inspektionen wurden in den siebziger Jahren von IBM eingeführt und sollen Zuverlässigkeit, Benutzbarkeit und Änderbarkeit einer Software verbessern [10].

In dieser Arbeit konzentrieren wir uns auf die Zuverlässigkeitsverbesserung, die durch eine Anwendung von Inspektion erreicht werden kann.

Eine Inspektion ist ein Kontrolle von einem Code, durchgeführt von Experten.

Die Experten haben als Hilfe eine Inspektion-Checkliste, um die einzelnen Schritte bei der Softwarekontrolle zu verfolgen. Die Experten untersuchen erst einzeln den Code und schreiben sich die gefundenen Fehler auf. Weiterhin notieren sie natürlich die Zeit, die sie für Ihre Untersuchung gebraucht haben.

Nachdem jeder Experte den Code untersucht hat, findet ein Treffen aller Experten statt, um gemeinsam die gefundenen Fehler zu besprechen.

Als Ergebnis der Inspektion, bekommt man eine Liste von Fehlern, die eingestuft sind in „Improvement“; Verbesserungsvorschlag, „Minor“; leichter Fehler und „Major“ schwerer Fehler.

3.1.2. Black-Box Tests

Black-Box Tests ist die Intuitivste der ausgewählten Methoden, denn bei diesem Testverfahren werden die Testdaten ausschließlich auf Basis der Spezifikation oder der Programmbeschreibung ermittelt.

Bei den Black-Box Test, auch funktionale Tests genannt, stellt der Entwickler Testfälle zusammen und berücksichtigt dabei, nach [11] Folgendes:

- **Bildung von Äquivalenzklassen** (Diese Klassen haben nichts mit Klassen im Sinne der OOP zu tun.) Das Programm reagiert bei allen Werten aus einer definierten Äquivalenzklasse gleich. Funktioniert das Programm mit einem Wert aus der Äquivalenzklasse fehlerfrei, so funktioniert es mit allen anderen Werten aus dieser Klasse ebenfalls korrekt. Durch dieses Verfahren lässt sich die Anzahl der erforderlichen Testfälle deutlich einschränken. (So können z.B. bei Notenpunkten die Äquivalenzklassen "*zu klein*" für alle Werte < 0 , "*korrekt*" für alle Werte im Bereich 0..15 und "*zu groß*" gebildet werden. Die jeweilige Programmfunktion kann nun mit drei repräsentativen Werten wie -2 , 5 und 20 hinreichend getestet werden.
- **Grenzwertanalyse**
Dieses Verfahren setzt voraus, dass die Werte innerhalb einer Äquivalenzklasse sinnvoll geordnet werden können, z.B. aufsteigend, absteigend, nach dem Wert oder der Zeit. Dies geht bei Mengen, Preisen etc., aber z.B. nicht bei Daten wie "Eigenschaften" oder "Kennbuchstaben". Bei der Grenzwertanalyse wird nicht irgendein beliebiger Wert aus einer Äquivalenzklasse, *sondern gezielt Randwerte* getestet. Erfahrungsgemäß tauchen hier am häufigsten Ausfälle auf. Im Gegensatz zur Äquivalenzklassenbildung kann man bei der Grenzwertanalyse auch aus der

Betrachtung der Ausgabewerte Testdaten ableiten. Beim praktischen Test wird man sowohl aus dem gültigen, wie auch aus dem ungültigen Wertebereich einen möglichst dicht an der jeweiligen Grenze liegenden Wert testen. Als Grenzwerte eignen sich Randwerte von Gültigkeitsintervallen, Maxima, Minima, Ausnahme- und Fehlerfälle, wie falsche Eingabezeichen oder Wert außerhalb des Wertebereiches.

- **Intuitive Testfallermittlung**

Besonders kritische Fälle aus der Erfahrung miteinbeziehen wie: extrem große oder kleine Werte, wirre Eingabefolgen, mit der Hand beliebig über die Tastatur gleiten oder absichtliche Fehlbedienungen.

3.1.3. Glas-Box Tests

Glas-Box Tests basieren auf dem Quellcode und versuchen die Möglichkeiten einer Software zu bedienen, aus dem Code zu extrahieren und sie in Testfälle umzusetzen.

Bei Glas-Box Tests werden die Testdaten nach [11] so gewählt, dass:

- jede Anweisung mindestens einmal ausgeführt wird. Dabei gilt eine Auswahl oder Fallunterscheidung insgesamt als eine Anweisung.
- jeder Programmzweig mindestens einmal durchlaufen werden muss. (Also jede Schleife wenigstens 1x durchlaufen werden und jede Auswahl wenigstens 1x im True- oder False-Teil durchlaufen werden.)
- jede Möglichkeit, die in einer Bedingung auftreten kann, mindestens einmal getestet werden muss. (Also alle einer Teile einer Auswahl, True- und False-Teil.)
- ein Pfad einen möglichen Weg durch das Programm vom Anfang bis zu Ende beschreibt. Jeder mögliche (denkbare !!) Pfad muss einmal durchlaufen werden, also auch jede denkbare Ereignisreihenfolge.

Es ist klar, dass schon bei einem mittleren Programm die möglichen Pfade nicht mehr vollständig durchlaufen werden können, da sie einfach zu viele sind.

Der Entwickler sollte die Anzahl der getesteten Pfade verringern, indem er z.B. triviale Fälle ignoriert, oder Pfade, die eine ähnliche bzw. gleiche Funktionalität durchführen, nur einmal testet. Man stellt fest, dass es reicht jeden Pfad des Programms durchzulaufen, um jede Möglichkeit durchlaufen zu haben, und diese Relation gilt genauso, für die weiteren zwei Testmöglichkeiten, also es reicht jede Möglichkeit, die in einer Bedingung auftreten kann durchzulaufen, um jeden Programmzweig durchlaufen zu haben. Die Vier genannte Punkte überdecken sich.

3.2. Repräsentative Tests

Repräsentative Tests richten sich nach dem Nutzungsprofil der Software. Es werden keine Ausnahme- und Grenzfälle miteinbezogen.

Es wird lediglich versucht, einen realen Ablauf der Software zu testen, gemäß ihrer spezifizierten Benutzung.

Deswegen bieten sich repräsentative Tests wunderbar an, um die Sicherheit und Zuverlässigkeit einer Software am Endstand zu messen. Sie sind aber nicht so genau und pedantisch wie die gerichteten Test, deswegen bieten sich die gerichteten Tests besser für Anwendungen an, die die Sicherheit und Zuverlässigkeit in der Entwicklungs- und Testphase einer Software verbessern.

Bei erster Betrachtung scheinen die repräsentativen Tests eine Untermenge von den Black-Box Tests zu sein, denn die Black-Box Tests versuchen die Spezifikation zu untersuchen und sollten alle Fälle abdecken, die im realen Einsatz der Software vorkommen könnten.

Es stellt sich natürlich die Frage, ob es einen kleinen Vorsprung für die Black-Box Methode gibt, da das Modell, das zur Messung benutzt wird, indirekt auf dieser Methode basiert. im Kapitel 6 wird diese Frage beantwortet.

3.3. Das Experiment

Diese Arbeit soll drei Sicherheit- und Zuverlässigkeitsmethoden vergleichen:

- **Inspektionen**
- **Black-Box Tests**
- **Glas-Box Tests**

Der Vergleich richtet sich nicht nur nach der Anzahl oder Art der gefundenen Fehler, sondern nachdem, welche Fehler überhaupt gefunden und behoben wurden. Denn nicht alle Fehler führen zum Auftreten eines Ausfalls.

Statistisch haben nur 33% aller Ausfälle ein MTTF größer als 5.000 Jahre [5].

Und deswegen sieht der Aufbau unseres Experimentes folgendermaßen aus:

- Erstens wird ein Entwicklungsstand angestrebt, der die Anwendung von Sicherheits- und Zuverlässigkeitsmethoden zulässt.
- Dieser Softwarestand wird „eingefroren“ und wird „Version 0.1“ genannt.
- Auf einer Kopie von „Version 0.1“ wird die Inspektion Methode angewandt, das Ergebnis nennen wir „Version 1.0“.
- Auf einer Kopie von „Version 0.1“ wird die Black-Box Methode angewandt, das Ergebnis nennen wir „Version 2.0“.
- Auf einer Kopie von „Version 0.1“ wird die Glas-Box Methode angewandt, das Ergebnis nennen wir „Version 3.0“.
- Jetzt wird die Zuverlässigkeit und Sicherheit bei allen vier Versionen gemessen und verglichen.

Für die Zuverlässigkeitsmessung benutzen wir das MTTF Maß. Es werden im Laufe des Experiments die TBF aufgenommen, damit wir am Ende den Mittelwert, das ist der MTTF, ziehen können..

Wie im Kapitel 2.3 erläutert, wird für die Sicherheitsmessung die Summe aller Schweregrade der aufgetretenen Ausfälle benutzt.

4. Fallstudie „Verkehrsleittechnik“

4.1. Beschreibung des technischen Systems

Dieser Kapitel basiert auf [7] und gibt eine Referenzfallstudie zurück.

Diese Referenzfallstudie befasst sich mit dem verkehrsleittechnischen Prozess der Steuerung und der Sicherung einer Fahrt eines Zuges auf einem dem entsprechenden Fahrweg. Dies geschieht auf der Basis des funkbasierten Fahrbetriebs. Die Studie beschränkt sich unter dem Gesichtspunkt der Steuerung und der Sicherung zustandsvariabler Fahrweegelemente (Bahnübergänge und Weichen) auf die hierfür interessierenden Teilsysteme, Funktionen und Schnittstellen. Der funkbasierte Fahrbetrieb lässt sich kurz und knapp wie folgt beschreiben: Der Fahrzeugrechner auf dem Schienenfahrzeug und die Leiteinrichtungen an der Strecke, d.h. die auf die verschiedenen Prozessobjekte verteilte Leittechnik, kommunizieren miteinander per Mobilfunktechnologie (GSM-R) und tauschen so Daten über die Anforderungen des Fahrweges, die Erteilung einer Fahrerlaubnis und für sonstige Kommunikationszwecke aus. Zentrale Stellwerke zur Fahrweegeinstellung und -sicherung werden bei dieser Betriebsverfahrensart nicht benötigt. Mit dem Wegfallen dieser ergibt sich das Problem der Steuerung und Sicherung von Bahnübergängen und Weichen.

Der Zug, der sich der Weiche oder dem Bahnübergang nähert, erhält über eine im Gleis liegende Balise Kenntnis vom vorausliegenden Gefahrenpunkt, seiner Art, seiner Entfernung und ggf. weiterer Variablen.

Entnimmt der Fahrzeugrechner aus dem Streckenatlas, dass die gerade überfahrene Balise ein zustandsvariables Fahrweegelement ankündigt, so sendet er an die entsprechende Elementsteuerung rechtzeitig einen Stellbefehl.

Der Fahrzeugrechner berechnet unmittelbar nach Bekanntwerden des Gefahrenpunktes eine neue Bremskurve, so dass das Fahrzeug im Zweifelsfall durch eine Zwangsbremmung vor dem Gefahrenpunkt sicher zum Stehen gebracht wird. Bei einem Bahnübergang wird einige Zeit vor dem Schließen der Schranken in der Regel eine Signal- bzw. Lichtzeichenanlage eingeschaltet. Sobald dem Fahrzeugrechner der gewünschte Elementstatus gemeldet wurde, nimmt dieser den Gefahrenpunkt aus

der Überwachung, setzt die sicherheitshalber eingestellte Bremskurve außer Kraft und überwacht wieder das ursprüngliche Geschwindigkeitsprofil. Der Fahrer braucht bei systemgesteuerten Bahnübergängen oder Weichen im Regelbetrieb keine Bedienhandlungen durchführen.

Im Störfall (z.B. bei Störung einer fahrzeug- bzw. streckenseitigen Leiteinrichtung) wird das Fahrzeug über die im Streckenatlas hinterlegte Bremskurve vor dem Gefahrenpunkt zum Halt geführt. Der Fahrzeugführer muss nach erfolgtem Halt, die auf seinem Display angezeigten Bedienanweisungen, die für die ordnungsgemäße Einstellung und Sicherung des zustandsvariablen Fahrweegelementes notwendig sind, durchführen. Nach der erfolgreichen Ausführung und Quittierung am Display wird der Bremsvorgang aufgehoben und die Fahrt kann fortgesetzt werden.

Das Fahrzeuggerät ermittelt bei jeder Überfahrt einer Balise, ob ein Bahnübergang oder eine Weiche naht. Dabei können im Streckenatlas einer Balise mehrere Bahnübergänge und/oder Weichen zugeordnet sein. Nähert sich der Zug einem oder mehreren zustandsvariablen Fahrweegelementen, so setzt das Fahrzeuggerät an den entsprechenden Gefahrenpunkten die Zielgeschwindigkeit "Null", was zu einer sofortigen Neuberechnung der Bremskurve führt.

Nach Überfahrt einer entsprechenden Balise bestimmt das Fahrzeuggerät den optimalen Zeitpunkt zum Absetzen des entsprechenden Elementstellbefehls über die fahrzeugseitige Funkeinrichtung.

Die Überwachung und Regelung/Einhaltung der Ist-Geschwindigkeit mit der Bremskurve erfolgt kontinuierlich.

Ist das Fahrzeug vor einem ungesicherten Gefahrenpunkt bis zum Halt gebremst worden, so wird die Fahrsperrung aktiviert. Sie bleibt so lange aktiviert, bis der Gefahrenpunkt aufgehoben wird: entweder automatisch durch den Empfang einer entsprechenden Elementstatusmeldung von der Steuerung des zugehörigen zustandsvariablen Fahrweegelements oder bei manueller Quittierung des Ordnungszustandes durch den Triebfahrzeugführer am Display.

Empfängt das Fahrzeuggerät von der Steuerung des zustandsvariablen Fahrweegelements die Meldung über den sicheren Elementstatus, so kann der Gefahrenpunkt aus der Bremskurvenüberwachung herausgenommen werden. Die

Zielgeschwindigkeit "Null" vor dem Gefahrenpunkt wird aufgehoben und die Bremskurve neu berechnet. Liegt die Ist-Geschwindigkeit zuvor über und danach unter der Bremskurve, so wird die Bremsung aufgehoben, bzw. bei einem bis zum Halt zwangsgebremstem Zug wird die Fahrsperre gelöst.

Die Bremskurve wird bei der zulässigen Geschwindigkeit am Gefahrenpunkt jeweils neu berechnet.

4.2. Beschreibung der Systemrealisierung

4.2.1. Die Simulation

Um die gewünschte Testumgebung zu rekonstruieren ist es schwierig, ein reales System herzunehmen, da die benötigten Komponenten nicht unbedingt jedem zur Verfügung stehen: Züge, Bahnstrecken usw., deshalb ist hier ein Software-Simulation notwendig.

Die Simulation soll so weit wie möglich, das in der Referenzfallstudie spezifizierte System abbilden, um die Realitätsnähe nicht ganz zu verlieren.

Die Referenzfallstudie geht von der reibungslosen Funktionalität des Funksystems aus, was die Implementierung der Software-Simulation ein wenig erleichtert, da keine Ausfälle im Funkbetrieb o.ä. simuliert werden müssen.

Die Simulation soll jedoch dem Benutzer die Möglichkeit geben, verschiedene Bahnstrecken aufzustellen und Hindernisse beliebig auf diese Strecken zu stellen. Der Benutzer soll auch die Möglichkeit haben, Züge auf bestimmten Strecken zu ausgewählten Zeitpunkten starten zu lassen. Am wichtigsten jedoch ist, dass der Benutzer klar erkennen soll, wann es zu einem Ausfall kommt.

Die Software-Simulation ist in Java realisiert, die Visualisierung erfolgt als Applet.

Auf dem gelben Hintergrund sind die Bahnstrecken als Linien zu sehen, die Bahnübergänge sind als rote Kreuz, die Weichen als rote Rechtecke und die Balisen als rote Dreiecke dargestellt.

Rechts oben befindet sich eine Dropdown-Liste mit den bestehenden Bahnstrecken, darunter ein "Add"-Button.

Mit dem Drücken des "Add"-Buttons startet sofort ein Zug auf dem in o. g. Dropdown-Liste ausgewählten Strecke.

Es gibt jedoch die Möglichkeit , mehrere Züge zu bestimmten Zeiten starten zu lassen, indem man in der Textarea ein Liste von auszuführenden Befehlen eingibt und anschließend auf dem "Go" -Button drückt.

Die Eingabe der Befehle soll in folgender Form erfolgen:

- jeder Befehl in einer neuen Zeile
- jede Zeile beinhaltet den Startpunkt des Zuges, z. B. A für die Strecke A->B

- Leerzeichen, dann die Verzögerungszeit vor der Ausführung von diesem Befehl in Millisekunden.

Die tatsächliche Verzögerungszeit ergibt sich aus der Summe der vorausgegangenen Verzögerungszeiten.

Hier ein Beispiel für die Eingabe:

```
A 1000  
C 1000  
A 0  
C 2000
```

Diese Befehle würden, angenommen der Zeitpunkt des Drückens des Button "Go" sei 0, Folgendes bewirken,:

Zeitpunkt 1 Sekunde: ein Zug startet von A;

Zeitpunkt 2 Sekunden: ein Zug startet von C;

Zeitpunkt 2 Sekunden: ein Zug startet von A;

Zeitpunkt 4 Sekunden: ein Zug startet von C;

Die Züge werden als schwarze Kreise dargestellt, und können in ihren Bewegungen verfolgt werden.

4.2.2. Die Implementierung

Bei der Implementierung der Software-Simulation, wird an aller erster Stelle versucht eine spezifikationsnahe Umgebung zu schaffen, um ein objektives Prüfen der Software zu ermöglichen. Es wird weniger Wert auf die Details gelegt, jedoch sind die Hauptzüge des spezifizierten Systems eingehalten worden.

Dies kann man vor allem bei der Implementierung der Steuerung des Funkbetriebes feststellen, da die Inhalte einer Nachricht nicht genauso übermittelt werden, wie sie im Dokument der Referenzfallstudie beschrieben sind, sondern die Nachricht dieselbe Botschaft überbringt, z.B. "Hindernis voraus" oder "Bahnübergang ist frei".

Im Weiteren wird mit einer Einheit, eine Streckeneinheit, bzw. hier in der Software-Simulation ein Pixel gemeint. Mit einem Takt ist ein Thread-Lauf gemeint, also in dieser Software-Simulation 80 Millisekunden.

Messaging :

Die Steuerung des Funkbetriebs ist im Package messaging zu finden; die Klasse Message implementiert ein Nachrichtenobjekt.

Eine Nachricht hat folgende Eigenschaften:

- eine eindeutige ID
- einen Nachrichtentyp
- eine Referenz auf dem Sender
- ggf. eine Bremskurve
- ein zu übergebender Status

Die Methode 'sendMessage()' schickt diese Nachricht an dem Empfänger, indem sie die jeweilige Nachricht an der Nachrichtenwarteschlange des Empfängerobjektes anreicht.

Es gibt drei Nachrichtentypen, hier die Schlüssel dafür:

- 0 ist eine Warnung vor einem kommenden Hindernis.
- 1 ist eine Aufforderung ein Hindernis (Bahnübergang oder Weiche) zu schließen, also keine Überquerung mehr möglich machen, bzw. ein Fahrzeugobjekt in dem Status "unlocked", also bereit, zu setzen.
- 2 ist eine Aufforderung ein Hindernis (Bahnübergang oder Weiche) zu öffnen, also Überquerungen wieder möglich machen.

Die Nachrichtentypen werden von den jeweiligen Senderobjekten festgelegt, je nach Situation.

Jede Nachricht beinhaltet nicht nur eine Referenz auf dem Objekt, von dem die Nachricht stammt, sondern auch einen Schlüssel, der besagt, was für ein Objekttyp der Sender ist:

- 0 der Sender ist eine Balise; Balisen verschicken Nachrichten ausschließlich an Fahrzeugobjekte als Warnung vor einem baldigen Hindernis. Diese Nachricht beinhaltet ebenso eine Bremskurve, die das Fahrzeug zum rechtzeitigen Stillstand vor dem Hindernis bringen soll. Das Fahrzeug bekommt den Zustand "locked", also nicht bereit, nachdem es eine solche Nachricht empfängt.

- 1 der Sender ist ein Bahnübergang; Bahnübergänge verschicken Nachrichten ausschließlich an Fahrzeugobjekte als Benachrichtigung, dass es möglich ist den Bahnübergang zu überqueren.
- 2 der Sender ist eine Weiche; Weichen verschicken Nachrichten ausschließlich an Fahrzeugobjekte als Benachrichtigung, dass es möglich ist, die Weiche zu überqueren.
- 3 der Sender ist ein Fahrzeugobjekt; Fahrzeugobjekte verschicken Nachrichten entweder an Weichen oder an Bahnübergänge, um sich für ein Überquerung anzumelden.

Engines :

Die Engines sind Systemkomponenten, also Balisen, Bahnübergänge, Weichen und Züge.

Diese sind in dem Package engine implementiert.

In folgendem Diagramm sind die möglichen Nachrichtentypen dargestellt, wie sie zwischen die verschiedenen Software-Komponenten versendet werden können.

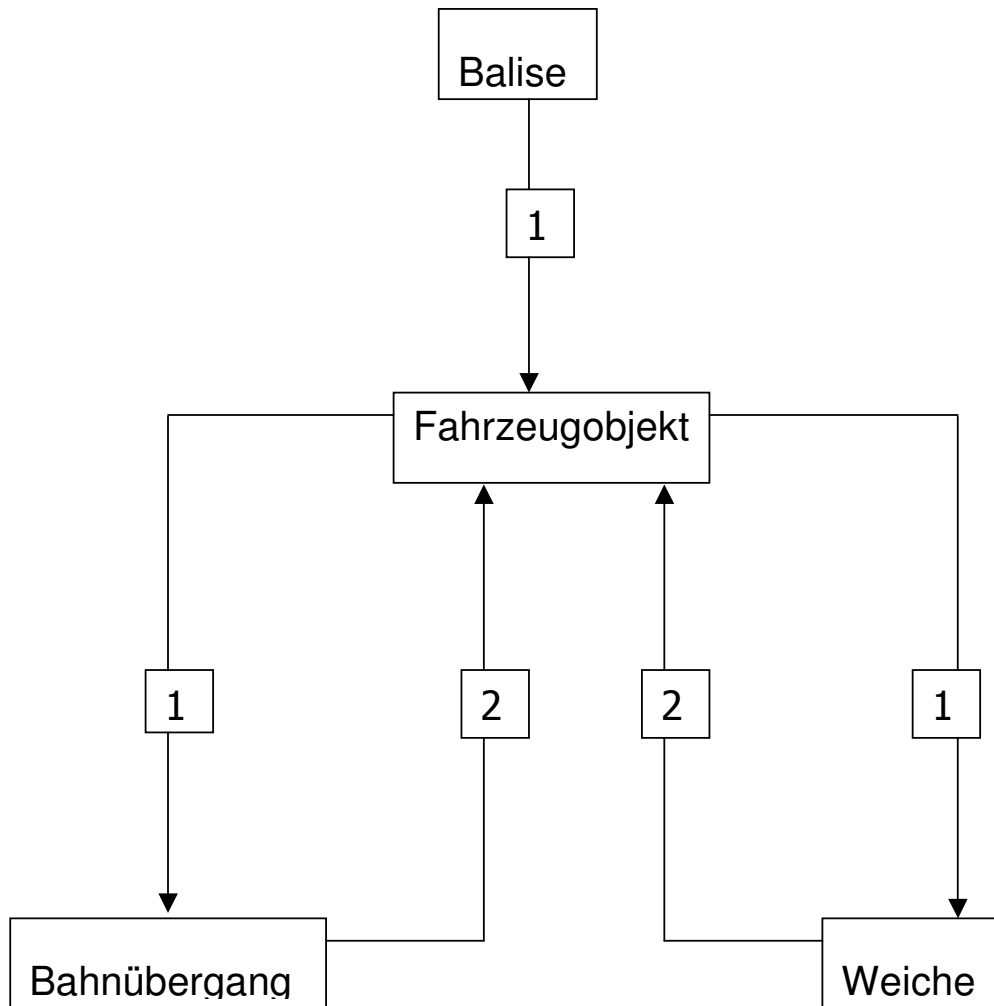


Abb. 1 : Mögliche Nachrichtentypen zwischen den Systemkomponenten

Eine Engine hat folgende Eigenschaften:

- eine eindeutige ID
- eine Position auf einer bestimmten Strecke im Streckenatlas

Jede Engine ist ein Thread, das alle 80 ms läuft und bestimmte Zustände prüft und dementsprechend Aktionen durchführt.

Es gibt vier verschiedene Engines, die im Folgenden näher beschrieben werden.

Balise-Engine:

Eine Balise hat folgende Eigenschaften:

- eine eindeutige ID
- eine Position auf einer bestimmten Strecke im Streckenatlas
- eine Nachricht, die jedem Zug zugeschickt wird, der diese Balise überquert

Die Balise versucht bei jedem Lauf herauszufinden, welche Züge in der Zwischenzeit die Balise überquert haben, um diesen Zügen die Balise-spezifische Nachricht zu senden.

Der Balisen-Thread merkt sich, welche Züge beim letzten Lauf auf der selben Strecke fahren als seine Strecke und speichert deren Positionen. Beim nächsten Lauf prüft der Thread ob die aktuelle Position eines Zuges nach der Position dieser Balise liegt und die zuletzt gespeicherte Position danach liegt. In diesem Fall würde es bedeuten, dass der Zug gerade über die Balise gefahren ist und dementsprechend eine Nachricht bekommen sollte.

Die von der Balise gesendete Nachricht beinhaltet ebenfalls eine Bremskurve.

Die Bremskurve ist eine Liste von Geschwindigkeiten, die der Zug einhalten musste, um die Geschwindigkeit 0 vor dem Hindernis zu erreichen.

Die Bremskurve wird folgendermaßen berechnet:

Wenn V_0 die Zuggeschwindigkeit zum Zeitpunkt des Überquerens der Balise ist und N_0 die Entfernung bis zum Hindernis ist (im konkreten Fall wird immer 7 Einheiten vor dem Hindernis ein vollständiger Stillstand angestrebt) dann wird nach jeder Einheit um V_0/N_0 gebremst, d.h. nach einer Einheit ist die neue Geschwindigkeit

$V_1 = V_0 - V_0/N_0$, und nach 2 Einheiten $V_2 = V_0 - V_0/N_2$, bis zum Stillstand

$V_{N-1} = V_0 - V_0/1$.

Die Bremskurve ist eine Liste von Geschwindigkeiten, die an bestimmten Positionen erreicht werden sollen und die das Fahrzeug einhalten soll, damit es vor dem Hindernis zum Stillstand kommen kann.

Bahnübergang-Engine :

Ein Bahnübergang hat folgende Eigenschaften:

- eine eindeutige ID
- eine Position auf einer bestimmten Strecke im Streckenatlas
- eine Nachrichtenwarteschlange auf der die empfangenen Nachrichten angereicht werden.
- einen Status, der besagt in welchem Zustand der Bahnübergang sich gerade befindet (geöffnet, also Überquerung möglich, oder geschlossen: keine Überquerung möglich)

Ein Bahnübergang prüft bei jedem Lauf, ob eine Nachricht in der Nachrichtenwarteschlange vorliegt.

Falls die Nachrichtenwarteschlange nicht leer ist, wird geprüft, ob der Bahnübergang geöffnet ist.

Falls der Bahnübergang geschlossen ist, wird solange gewartet, bis er wieder geöffnet ist.

Falls der Bahnübergang geöffnet ist, wird zuerst der Bahnübergang geschlossen, damit ihn keine anderen Fahrzeuge mehr überqueren können, daraufhin wird eine Nachricht an das Fahrzeug zugeschickt, das die gerade bearbeitete Nachricht gesendet hat.

Diese Nachricht erlaubt dem Fahrzeug diesen Bahnübergang zu überqueren und sichert zu, dass dieser Übergang in der Zwischenzeit für alle andere Fahrzeuge geschlossen bleiben wird.

Nachdem das Fahrzeug benachrichtigt wurde, dass es durchfahren darf, wird die zuletzt bearbeitete Nachricht aus der Nachrichtenwarteschlange des Bahnübergangs entfernt.

15 Einheiten nach dem Überqueren des Bahnübergangs, schaltet das Fahrzeugobjekt den Bahnübergang in den Zustand "unlocked", also geöffnet. Dieser

Vorgang ersetzt den Schalter, den es im realen System gibt, um festzustellen, dass ein Zug den Bahnübergang vollständig überquert hat.

Weiche :

Eine Weiche hat folgende Eigenschaften:

- eine eindeutige ID
- eine Liste von Positionen im Streckenatlas. Da die Weiche auf mindestens zwei Strecken liegt, ist hier eine Liste von jeweils die Position dieser Weiche auf diesen Strecken
- eine Nachrichtenwarteschlange auf der die empfangenen Nachrichten angereiht werden.
- ein Status, der besagt in welchem Zustand die Weiche sich gerade befindet (geöffnet, also Überquerung möglich, oder geschlossen: keine Überquerung möglich)

Eine Weiche prüft bei jedem Lauf, ob eine Nachricht in der Nachrichtenwarteschlange vorliegt.

Falls die Nachrichtenwarteschlange nicht leer ist, wird geprüft, ob die Weiche geöffnet ist.

Falls die Weiche geschlossen ist, wird solange gewartet bis sie wieder geöffnet ist.

Falls die Weiche geöffnet ist, wird zuerst die Weiche geschlossen, damit keine andere Fahrzeuge sie mehr überqueren, dann wird eine Nachricht an das Fahrzeug geschickt, das die gerade bearbeitete Nachricht gesendet hat.

Diese Nachricht erlaubt dem Fahrzeug diese Weiche zu überqueren, und sichert zu, dass diese Weiche in der Zwischenzeit für alle andere Fahrzeuge geschlossen bleiben wird.

Nachdem das Fahrzeug benachrichtigt wird, dass es passieren darf, wird die zuletzt bearbeitete Nachricht aus der Nachrichtenwarteschlange der Weiche entfernt.

15 Einheiten nach dem Überqueren der Weiche, Schalter der Fahrzeug Objekt die Weiche in dem Zustand "unlocked", also geöffnet, um. Dieser Vorgang ersetzt den Schalter, den es im realen System gibt, um festzustellen, dass ein Zug die Weiche vollständig überquert hat.

Fahrzeugobjekt-Engine :

Ein Fahrzeug hat folgende Eigenschaften:

- eine eindeutige ID
- eine Nachrichtenwarteschlange auf der die empfangenen Nachrichten angereicht werden.
- ein Status, der besagt in welchem Zustand sich der Fahrzeugobjekt befindet.
- die aktuelle Geschwindigkeit
- die aktuelle Position auf einer bestimmten Strecke im Streckenatlas
- eine Bremskurve, die aber nur benötigt wird, wenn ein Hindernis droht

Die Fahrzeug-Engine berechnet bei jedem Lauf die neue Position des Fahrzeugobjektes, indem es die Taktlänge mit der Geschwindigkeit des Fahrzeugs multipliziert.

Die Geschwindigkeit des Fahrzeugobjektes hängt vom Status des Zuges ab. Falls der Zug den Status "locked" hat, bedeutet dies, dass der Zug auf ein Hindernis zusteuert und eine Nachricht diesbezüglich bereits erhalten hat. Die Geschwindigkeit des Fahrzeugobjektes wird in diesem Fall der Bremskurve entnommen.

Falls das Fahrzeugobjekt den Status "unlocked" hat, ist die Geschwindigkeit einfach aus der Instanzvariablen zu entnehmen.

Das Fahrzeugobjekt prüft nach jedem Takt, ob die Nachrichtenwarteschlange leer ist. Falls eine Nachricht vorliegt, reagiert das Fahrzeugobjekt folgendermaßen:

Falls die Nachricht vom Typ 0 ist, also eine Warnung von einem voraussichtlichen Hindernis, wird geprüft ob das Fahrzeugobjekt nicht bereits im Zustand "locked" ist. Ist dies nicht der Fall, dann wird der Status des Fahrzeugobjektes sofort in "locked" umgeschaltet und der Zug schickt eine Nachricht an das Hindernis, damit das Hindernis eine Überquerung ermöglicht.

Falls die Nachricht vom Typ 2, also eine Aufforderung, das Fahrzeugobjekt in den Status "unlocked" zu setzen ist, bedeutet es, dass das Hindernis einen freien Durchgang für dieses Fahrzeug bescheinigt. In diesem Fall wird die Bremskurve invalidiert und das Fahrzeug ist wieder im bereiten Zustand und kann weitere Nachrichten bearbeiten.

Streckenatlas :

Um die späteren Tests leichter durchführen zu können, benötigen wir einen dynamischen Streckenatlas der schnell und einfach zu ändern ist.

Um diesen Zweck zu erfüllen, hat diese Softwaresimulation eine Eingabedatei, die die Struktur des Streckenatlas festlegt.

In der Eingabedatei sind nur die einzelnen Strecken und die Bahnübergänge anzugeben, da die Weichen bei jeder Überschneidung von Strecken automatisch berechnet werden können. Die Balisen werden ebenfalls automatisch berechnet, da sie immer vor den Hindernissen positioniert werden müssen.

Die Struktur der Eingabedatei muss wie folgt aussehen:

Der Schlüsselwort "path" sagt an, dass die Angabe der Strecken beginnt; in jeder Zeile wird eine neue Strecke definiert.

Die Zeile für eine Strecke fängt immer mit dem Startnamen dieser Strecke an, danach ein Leerzeichen, dann Komma, und Leerzeichengetrennt die einzelnen Segmente dieser Strecke in der Form x_1,y_1,x_2,y_2 .

x_1 und y_1 sind die Koordinaten des ersten Punktes aus dem Segment und x_2,y_2 sind die Koordinaten des zweiten Punktes.

Nach der Angabe aller Segmente der Strecke kommt der Endname der Strecke

Der Schlüsselwort "bridge" sagt an, dass die Angabe der Bahnübergänge beginnt; in jeder Zeile wird ein neuer Bahnübergang definiert.

Die Zeile für einen Bahnübergang besteht aus der Streckennummer, so wie diese Strecke in der Datei definiert wurde, und die Position des Bahnüberganges auf dieser Strecke.

Hier ein Beispiel für eine Eingabedatei:

```
path
A 100,20,100,500 100,500,500,300 500,300,500,450
500,450,600,450 B
C 20,100,200,100 200,100,500,60 500,60,600,60 600,60,400,400
400,400,600,400 D
E 50,300,300,500 300,500,300,100 300,100,600,100 F
bridge
0 100
```

- 1 780
- 1 80
- 2 780

Der spezifizierte Streckenatlas ist in der Klasse routeAtlas definiert und kennt außerdem noch folgende Klassen:

Streckenpunkt :

In der Klasse pathPoint ist die Struktur eines zweidimensionalen Punktes mit den Koordinaten x und y definiert.

Streckenposition :

In der Klasse pathPosition ist eine Position auf einer Strecke definiert.

Eine Streckenposition ist definiert durch eine Referenz auf einer bestimmten Strecke und einer Angabe der Position auf dieser Strecke.

Die Position muss aber in manchen Situationen im Koordinatensystem umgerechnet werden. Dafür sorgt die Methode "getPointByPathPosition()". Diese Methode wird z.B. gebraucht, wenn ein Objekt, das nur durch seine Position auf einer Strecke bekannt ist, visualisiert werden muss.

Streckensegment :

In der Klasse pathSegment ist die Struktur eines Segmentes definiert, also Koordinaten vom ersten Punkt x1,y1 und Koordinaten vom zweiten Punkt x2,y2.

Strecke :

In der Klasse routeAtlasPath ist die Struktur für eine Strecke definiert.

Eine Strecke besteht aus mehreren zusammenhängenden Segmenten. Sie hat einen Start- und Endnamen und eine Liste von allen Fahrzeugobjekten, die drauf fahren, sowie alle Balisen, Bahnübergänge und Weichen, die darauf liegen.

Bei jeder Definition einer neuen Strecke wird die Methode "registerSwitchesForPath(RouteAtlasPath rap, RouteAtlas ra)" aufgerufen. Diese Methode stellt alle Überschneidungen dieser Strecke mit der bereits in diesem Streckenatlas bestehenden Strecken fest und nimmt diese Überschneidungen als Weichen auf.

Die Methoden "getUniqueSwitchId()", "getUniqueBridgId()", "getUniqueBaliseId()" erzeugen eine eindeutige ID auf der aktuellen Strecke für eine Weiche, einen Bahnübergang bzw. eine Balise.

Streckenatlas :

In der Klasse routeAtlas ist die Struktur für einen Streckenatlas definiert.

Ein Streckenatlas besteht aus einer Liste von Strecken, einer Liste von Fahrzeugen, einer Liste von Weichen und einer Liste von Bahnübergängen. Dafür, dass der Streckenatlas zur Laufzeit aus einer Eingabedatei gebaut wird, sorgt die Methode "readRouteAtlasFromFile(File file)". Diese Methode liest die Eingabedatei ein und nimmt alle Strecken und Bahnübergänge in diesen Atlas auf. Die Balisen und Weichen werden automatisch gesetzt.

Die Visualisierung :

Die Software-Simulation wird als Java-Applet visualisiert.

Nachdem der Streckenatlas einlesen wurde, kann es angezeigt werden.

Die Fahrzeugobjekte aktualisieren nach jedem Takt ihre Position selbst, nachdem sie diese neue Position berechnet haben.

Die Bedienung ist in Kapitel 2 näher beschrieben.

In der Abbildung 2 ist ein Screenshot von der Oberfläche.

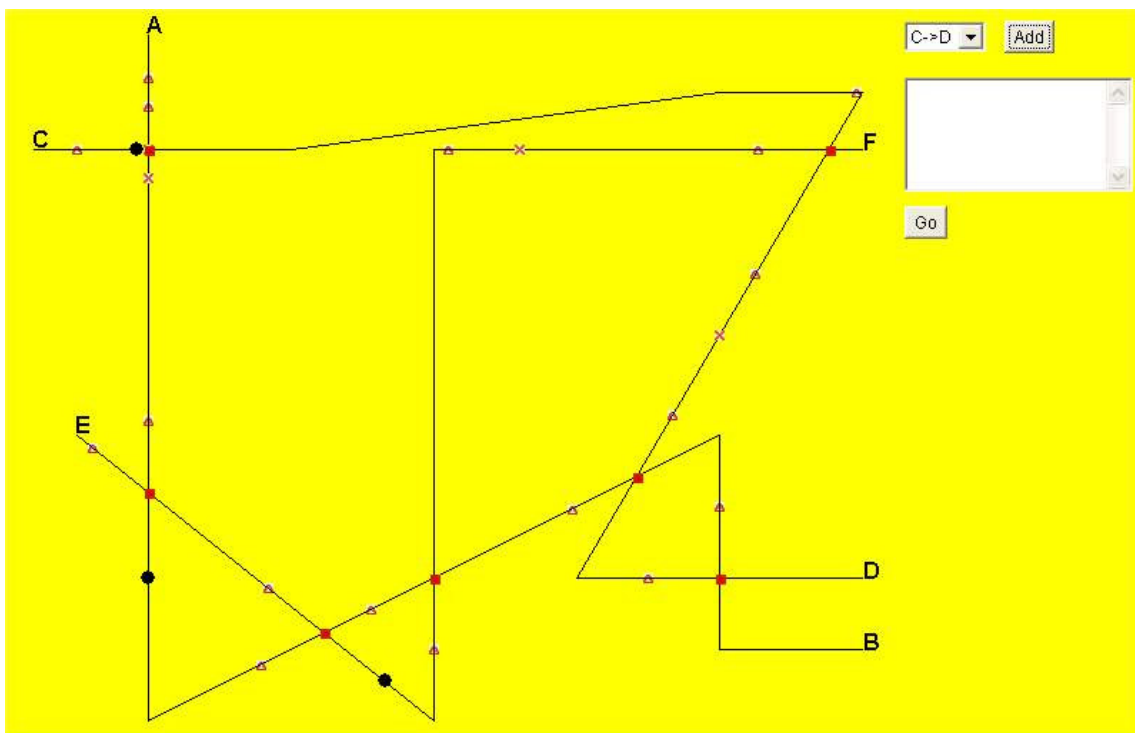


Abb. 2: Screenshot der Programmoberfläche

5. Methodenanwendung

5.1. Inspektion

Bei der Inspektion wurde ein Standard Java Inspektion Checklist benutzt [12]. Die Checklist wurde von drei Personen unabhängig voneinander durchgegangen und anschließend bei einer Sitzung besprochen.

Jeder „Inspektor“ notierte seine gefundene Fehler, um sie anschließend mit den Fehlern der anderen zu vergleichen.

Die Vorbereitung und Ausführung der Inspektion hat insgesamt 150 Minuten gedauert.

Die einzelnen Punkte der Inspektion sind:

1. Deklaration von Variablen, Attributen und Konstanten

- Werden variable und konstante Namen in Übereinstimmung mit der Namensgebung verwendet?
- Gibt es Variablen oder Attribute mit verwirrend ähnlichem Namen?
- Wird jede Variable und jedes Attribut richtig geschrieben?
- Wird jede Variable und jedes Attribut richtig initialisiert?
- Können irgendwelche globalen Variablen lokal deklariert werden?
- Werden alle For-Schleifensteuervariablen im Schleifen-Header deklariert?
- Gibt es literale Konstanten, die als Konstanten definiert werden sollen?
- Gibt es Variable oder Attribute, die Konstanten sein sollten?
- Gibt es Attribute, die lokale Variable sein sollten?
- Haben alle Attribute passende modifiers (privat, protected, public)?
- Gibt es statische Attribute, die nicht-statisch sein sollten oder umgekehrt?

2. Deklaration von Methoden

- Werden Methodennamen in Übereinstimmung mit der Namensgebung verwendet?
- Wird jeder Methodenparameterinhalt, der vor überprüft wird verwendet?

- Für jede Methode: Bringt sie den korrekten Wert an jedem Methodenrückkehrpunkt zurück?
- Haben alle Methoden passende modifiers (privat, protected, public)?
- Gibt es statische Methoden, die nicht-statisch sein sollten oder umgekehrt?

3. Deklaration von Klassen

- Hat jede Klasse passende Constructors und Destructors?
- Haben irgendwelche Unterklassen allgemeine Eigenschaften, die in den superclass sein sollten?
- Kann die Klassenhierarchie vereinfacht werden?

4. Daten Referenz

- Ist jeder Array-Indexlistenwert innerhalb der definierten Grenzen?
- Objekt- oder Array-Referenz: Ist der Wert sicher nicht null?

5. Berechnung / Numerik

- Gibt es irgendeine Berechnung mit Mischdatentypen?
- Ist Überlauf oder Unterlauf während einer Berechnung möglich?
- Für jeden Ausdruck mit mehr als einem Operator: Sind die Annahmen über die Auswertungsreihenfolge korrekt?
- Werden Klammern verwendet, um Mehrdeutigkeit zu vermeiden?

6. Vergleich

- Für jeden Booleschen Test: Wird der korrekte Zustand überprüft?
- Sind die Vergleichsoperatoren korrekt?
- Ist jeder Boolesche Ausdruck vereinfacht worden?
- Ist jeder Boolesche Ausdruck korrekt?
- Gibt es unsachgemäße und unbemerkte Nebenwirkungen eines Vergleiches?
- Hat man "&" unbeabsichtigt ausgetauscht mit "&&" oder "|" mit "||"?

7. Flow Control

- Für jede Schleife: Wird die beste Wahl des looping-constructs verwendet?
- Terminieren alle Schleifen?

- Wenn es mehrfache Ausgänge von einer Schleife gibt: Ist jeder Ausgang notwendig und richtig angefasst?
- Hat jede switch einen default Fall?
- Sind fehlende switch Fälle korrekt und mit einer Anmerkung markiert?
- Ist die Verschachtelung der Schleifen zu tief? Und ist sie korrekt?
- Kann irgendein verschachteltes if-Statement in ein switch-Statement umgewandelt werden?
- Sind null-bodied Steuerstrukturen korrekt und mit Klammern oder Anmerkungen gekennzeichnet ?
- Werden alle Exceptions passend angefasst?
- Terminiert jede Methode?

8. Input/Output

- Sind alle Dateien vor Gebrauch geöffnet worden?
- Sind die Attribute des input-object mit dem Gebrauch der Datei gleichbleibend?
- Sind alle Dateien geschlossener nach Gebrauch geschlossen worden?
- Gibt es Spelling oder grammatische Störungen in irgendeinem gedruckten oder angezeigten Text?
- Werden alle ioExceptions in einer angemessenen Weise angefasst?

9. Modul-Schnittstellen

- Sind die Zahl, die Reihenfolge, die Typen und die Werte Parametern in jedem Methodenaufruf in Übereinstimmung mit die benannten Definition der Methode?
- Stimmen die Werte in den Maßeinheiten (z.B., Zoll für Gelände)?
- Wird jeder benutzter Objekt oder Array, das benutzt wird richtig geändert?

10. Kommentare

- Hat jede Methode, Klasse und Datei einen passenden Kommentar?
- Hat jedes Attribut und jede Variablen und -Konstantendefinition einen Kommentar?
- Wird das zugrundeliegende Verhalten jeder Methode und Klasse in der normalen Sprache ausgedrückt?

- Ist der Kommentar für jede Methode und Klasse mit dem Verhalten der Methode oder der Klasse übereinstimmend?
- Passen die Kommentare und der Code zusammen?
- Helfen die Kommentare zum Verstehen des Codes?
- Gibt es genug Kommentare im Code?
- Gibt es zu viele Kommentare im Code?

11. Layout und packaging

- Wird ein Standardeinrückungs- und Layoutformat durchweg verwendet?
- Für jede Methode: Ist es nicht mehr als ungefähr 60 Zeilen lang?
- Für jedes kompilieren Modul: Ist nicht mehr als ungefähr 600 Zeilen lang?

12. Modularität

- Gibt es ein niedriges „Level Coupling“ zwischen Modulen (Methoden und Klassen)?
- Gibt es einen hohen Zusammenhang innerhalb jedes Moduls (Methoden oder Klassen)?
- Gibt es sich wiederholende Codes, die durch einen Anruf zu einer Methode ersetzt werden könnte, die das Verhalten des sich wiederholenden Codes liefert?
- Werden die Javaklassenbibliotheken verwendet, wo und wann es passt?

13. Speicherverbrauch

- Sind die Arrays groß genug?
- Werden Objekte und Arrays Referenzen auf Null gestellt, sobald der Gegenstand oder der Array nicht mehr benötigt wird?

14. Performance

- Können bessere Datenstrukturen oder effizientere Algorithmen angewandt werden?
- Sind die logischen Tests, die so geordnet werden, dass die häufig erfolgreichen und billigen Tests die kostspieligeren und weniger häufig erfolgreicheren Tests vorangehen?

- Können die Kosten der Neuberechnung eines Wertes durch er einmal berechnen und die Speicherung der Resultate verringert werden?
- Wird jedes Resultat, das berechnet und gespeichert, wirklich verwendet ?
- Kann eine Berechnung außerhalb einer Schleife verschoben werden?
- Gibt es Tests innerhalb einer Schleife, die nicht durchgeführt werden müssen?
- Kann eine kurze Schleife entrollt werden?
- Gibt es zwei Schleifen, die auf den gleichen Daten funktionieren, die in eine kombiniert werden können?
- Häufig sind die verwendeten erklärten Variablen Register?
- Sind die kurzen und allgemein benannten Methoden, die inline erklärt werden?

Nach der Anwendung diese Checkliste ergab sich eine Reihe von Fehlern.

Diese Fehler wurden in 3 Kategorien eingestuft:

- Schwere Fehler (Major Fault), sind in der Fehlerliste mit „Maj“ in der Spalte Schweregrad gekennzeichnet
- Leichte Fehler (Minor Fault), sind in der Fehlerliste mit „Min“ in der Spalte Schweregrad gekennzeichnet
- Verbesserungsvorschläge (Improvement) sind in der Fehlerliste mit „Imp.“ in der Spalte Schweregrad gekennzeichnet.

Klasse	Schweregrad	Fehler	Zeile
Message	Min	Kommentare fehlen	4
	Min	Instanzenvariablen sind alle public	15
	Min	Typisierung mangelhaft (Object)	23
	Min	Kommentare fehlen	39
	Maj	Id erzeugen unsicher	48
	Min	Kommentare fehlen	53
	Min	Default Fall fehlt bei der Switch	56
	Imp.	Oberklasse für switch und bridge statt Object	53
BaliseEngine	Min	Kommentare fehlen	1-6 und 15-20
	Min	Instanzenvariablen sind alle public	20
	Min	init Methode leer	31
	Min	Kommentare fehlen	36

	Min	Label überflüssig	42
	Maj	Exception Handling fehlt	82
	Min	Kommentare fehlen	86
Bridge	Min	Kommentare fehlen	10
	Min	Instanzenvariablen sind alle	24
	Imp.	Queue als Typ für message Typ	27
	Min	Init überflüssig	35
	Min	Kommentare fehlen	38
	Min	Status kann überschrieben werden	58
	Maj	Exception Handling fehlt	77
SwitchEngine	Min	Kommentare fehlen	10
	Min	Instanzenvariablen sind alle public	24
	Min	Init überflüssig	45
	Min	Status kann überschrieben werden	67
	Maj	Exception Handling fehlt	86
VehicleEngine	Min	Kommentare fehlen	14
	Min	Instanzenvariablen sind alle public	28
	Min	Init überflüssig	52
	Min	run Methode zu lang	57
	Maj	Default Fall fehlt bei der Switch	140
	Maj	Exception Handling fehlt	197
PathPoint	Min	Kommentare fehlen	1
	Min	Instanzenvariablen sind alle public	16
PathPosition	Min	Kommentare fehlen	1
	Min	Instanzenvariablen sind alle public	18
	Min	Variablendefinition überflüssig	26
	Min	Kommentare fehl am Platz	25
	Imp.	Klammerung wäre besser	67
PathSegment	Min	Kommentare fehlen	1
	Min	Instanzenvariablen sind alle public	15

RouteAtlasPath	Min	Kommentare fehlen	1
	Min	Instanzenvariablen sind alle public	24
	Min	Methode zu lang	44
	Min	Unvollständigkeit im Code	145
	Min	Assertion fehlt	216
RouteAtlas	Min	Kommentare fehlen	1
	Min	Instanzenvariablen sind alle public	27
	Min	Kommentare fehlen	30
Visualizate	Min	Kommentare fehlen	1
	Min	Instanzenvariablen sind alle public	37
	Maj	Exception Handling fehlt	51
	Imp.	Kommentar überflüssig	53
	Imp.	Kommentar nicht klar	56
	Min	Kommentare fehlen	80
	Min	Methode zu lang	80
	Imp.	Eingaben werden nicht validiert	176
	Maj	Exception Handling fehlt	197

Tabelle 1. : Ergebnistabelle der Inspektion

5.2. Black-Box Tests

Bei einem Black Box Test steht nur die Spezifikation des Moduls zur Verfügung, aus der die Tester die Testfälle generieren. Um alle Ausfälle zu finden, muss ein vollständiger Eingabetest durchgeführt werden. Das bedeutet, dass nicht nur alle zulässigen Eingaben getestet werden müssen, sondern alle möglichen Eingabedaten getestet werden müssen.

Die Vorbereitung und Ausführung der Black-Box Tests hat 80 Minuten gedauert.

Um die Tests besser beschreiben zu können werden folgende Schemata, als Komponentenzusammensetzung, definiert:

1. Eine Strecke „A“ mit einem Bahnübergang.
2. Eine Strecke „A“ und eine Strecke „C“ schneiden sich auf der Weiche „X“.
3. Wie Schema 2 nur auf der Strecke „A“ liegt ein Bahnübergang kurz vor der Weiche „X“.
4. Wie Schema 2 nur auf der Strecke „A“ liegt ein Bahnübergang kurz nach der Weiche „X“.
5. Wie Schema 3 nur auf der Strecke „C“ liegt ein Bahnübergang kurz vor der Weiche „X“.
6. Wie Schema 4 nur auf der Strecke „C“ liegt ein Bahnübergang kurz nach der Weiche „X“.
7. Wie Schema 3 nur auf der Strecke „C“ liegt ein Bahnübergang kurz nach der Weiche „X“.
8. Wie Schema 4 nur auf der Strecke „C“ liegt ein Bahnübergang kurz vor der Weiche „X“.

Tests auf Schema 1:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit geringen Abstand zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit geringen Abstand auf der Strecke „A“

Tests auf Schema 2:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit geringen Abstand zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit geringen Abstand zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
5. Zwei Züge 1 und 2 fahren mit geringen Abstand zueinander auf der Strecke „A“ und zwei weitere Züge 3 und 4 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
6. Drei Züge 1, 2 und 3 fahren mit geringem Abstand zueinander auf die Strecke „A“ und zwei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 3:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
5. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“ und zwei weitere Züge 3 und 4 fahren mit geringen Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
6. Drei Züge 1, 2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“ und drei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander

auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 4:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
5. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“ und zwei weitere Züge 3 und 4 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
6. Drei Züge 1, 2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“ und drei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 5:

1. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.
2. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“.
3. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden
4. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“, und zwei weitere Züge 3 und 4 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

5. Zwei Züge 1, 2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“, und drei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 6:

1. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.
2. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“.
3. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
4. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“, und zwei weitere Züge 3 und 4 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
5. Zwei Züge 1, 2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“, und zwei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 7:

1. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.
2. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“.
3. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden
4. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“, und zwei weitere Züge 3 und 4 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

5. Drei Züge 1, 2 und 3 fahren mit geringen Abstand zueinander auf der Strecke „A“ und drei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 8:

1. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.
2. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“.
3. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden
4. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“ und zwei weitere Züge 3 und 4 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
5. Drei Züge 1, 2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“ und drei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Nach der Ausführung der einzelnen Tests ergab sich eine Ausfallliste.

Der Schweregrad des Ausfalls entscheidet über die Sicherheit und ist in Stufen von 0-10 eingetragen, 0 bedeutet : es ist kein Ausfall bei dem Test aufgetreten, 10: bedeutet, es ist ein sehr schwerwiegender Ausfall aufgetreten.

Da auch die Zeit eine große Rolle bei der Auswertung der Ergebnisse spielt, und zwar nicht nur um den MTTF zu bestimmen, sondern auch um die Effizienz einer Methode festzustellen, wurde auch bei der Ausführung dieser Testsuite die Zeit gestoppt.

Die Ausführungszeit der Black-Box Tests betrug 80 Minuten.

Testschema	Testnummer	Schweregrad	Ausfall
Schema 1	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
Schema 2	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
6	10	Es wird keine Priorität bei der Überquerung gesetzt	
Schema 3	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	die vorderen Züge werden von hinteren überholt
6	10	Es wird keine Priorität bei der Überquerung gesetzt	
Schema 4	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	

	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt
Schema 5	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt
Schema 6	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt
Schema 7	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	

	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt
Schema 8	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt

Tabelle 2. : Ergebnistabelle der Black-Box Tests

5.3. Glas-Box Tests

Die Glas-Box Tests sind Test, die sich aus der Betrachtung des Quell-Codes und der Verfolgung der möglichen Zustandübergänge ergeben.

Im Gegensatz zum Black Box Test steht bei einem White Box Test dem Tester der komplette Code zur Verfügung. Mit dem White Box Test wird also die interne Struktur des Programms getestet. Um die ganze logische interne Struktur eines Programms zu testen, könnte man folgern, dass dies dadurch erreicht wird, indem alle Pfade eines Programms mit dem Testen durchlaufen werden.

Die Vorbereitung und Ausführung der Glas-Box Tests hat 150 Minuten gedauert.

Um die Tests besser beschreiben zu können werden folgende Schemata, als Komponentenzusammensetzung, definiert:

1. Eine Strecke „A“ mit einem Bahnübergang.
2. Eine Strecke „A“ und eine Strecke „C“ schneiden sich auf der Weiche „X“
3. Wie Schema 2 nur ein Bahnübergang liegt über der Weiche „X“.
4. Wie Schema 2 nur ein Bahnübergang liegt über der Balise der Weiche „X“.
5. Wie Schema 2 nur die Weiche „X“ liegt über der Balise von einem nach „X“ kommenden Bahnübergang

Tests auf Schema 1:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“

Tests auf Schema 2:

1. Zug 1 fährt auf der Strecke „A“.

2. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden
5. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“, und zwei weitere Züge 3 und 4 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
6. Drei Züge 1, 2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“, und drei weitere Züge 4, 5 und 6 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 3:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden
5. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“, und zwei weitere Züge 3 und 4 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
6. Drei Züge 1, 2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“, und drei weitere Züge 4, 5 und 6 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und

4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 4:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden
5. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“ und zwei weitere Züge 3 und 4 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
6. Drei Züge 1, 2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf die Strecke „A“, und drei weitere Züge 4, 5 und 6 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Tests auf Schema 5:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
5. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“, und zwei weitere Züge 3 und 4 fahren mit

einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

6. Drei Züge 1, 2 und 3 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“, und drei weitere Züge 4, 5 und 6 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Testschema	Testnummer	Schweregrad	Ausfall
Schema 1	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
Schema 2	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
6	10	Es wird keine Priorität bei der Überquerung gesetzt	
Schema 3	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten

	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt
Schema 4	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt
Schema 5	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Züge gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt

Tabelle 3. : Ergebnistabelle der Glas-Box Tests

5.4. Repräsentative Tests

Beim repräsentativen Testen werden alle Funktionen entsprechend der Häufigkeit, mit der sie später im Einsatz sein werden, getestet.

Die repräsentativen Tests versuchen die Gegebenheit wiederzustellen, deren das System in der Realität gesetzt wird. Hier wird nicht versucht Ausfälle zu finden und die Fehler, die dahinterstecken, zu beheben, bzw. das System auf Grenz- und Ausnahmefälle durchzutesten, sondern es wird lediglich versucht, das System unter die spezifizierte Umgebung laufen zu lassen.

Wie der Name „repräsentative“ es verrät, sollen diese Tests vertretbare Tests für die Realität sein, d.h. eine Darstellung von diesem System, wie es unter normalen Umständen benutzt wird.

Laut der obigen Definition sind die repräsentativen Test eine Untermenge von den Black-Box Tests, wobei die Grenz- und Ausnahmefälle nicht miteingeschlossen sind. Um die Tests besser beschreiben zu können werden folgende Schemata, als Komponentenzusammensetzung, definiert:

1. Eine Strecke „A“ mit einem Bahnübergang.
2. Eine Strecke „A“ und eine Strecke „C“ schneiden sich auf der Weiche „X“

Tests auf Schema 1:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.
3. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“

Tests auf Schema 2:

1. Zug 1 fährt auf der Strecke „A“.
2. Zwei Züge 1 und 2 fahren mit geringem Abstand zueinander auf der Strecke „A“.

3. Drei Züge 1,2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“.
4. Zwei Züge 1, auf der Strecke „A“, und 2, auf der Strecke „C“, fahren so, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
5. Zwei Züge 1 und 2 fahren mit einem Abstand kleiner als 15 Einheiten zueinander auf der Strecke „A“ und zwei weitere Züge 3 und 4 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 3 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.
6. Drei Züge 1, 2 und 3 fahren mit geringem Abstand zueinander auf der Strecke „A“ und drei weitere Züge 4, 5 und 6 fahren mit geringem Abstand zueinander auf der Strecke „C“; 1 und 4 sollen so fahren, dass sie zum selben Zeitpunkt die Weiche „X“ erreichen würden.

Um die Entwicklung der Sicherheit und Zuverlässigkeit nach der Anwendung jeder Methode messen zu können, werden die oben definierten „repräsentativen“ Tests auf vier Software-Versionen angewandt:

- **Version 0.1:** Der ursprüngliche Softwarestand, bevor irgendeine Methode zur Verbesserung der Sicherheit und Zuverlässigkeit benutzt wurde.
- **Version 1.0:** Der Softwarestand nach der Anwendung der „Inspektion“ Methode
- **Version 2.0:** Der Softwarestand nach der Anwendung der „Black-Box“ Methode
- **Version 3.0:** Der Softwarestand nach der Anwendung der „Glas-Box“ Methode

Analog zu den anderen Ergebnistabellen sind die auftretenden Ausfälle in Schweregrade zwischen 0 und 10 eingestuft worden. 0 bedeutet der Test ist ohne Ausfälle abgelaufen und 10 bedeute es ist ein sehr sicherheitskritischer Ausfall aufgetreten.

Die Ausführung aller nachfolgenden repräsentativen Tests hat jeweils 30 Minuten gedauert.

Die Ergebnisse dieser Tests sind in den nachfolgenden Tabellen ausgeführt:

Repräsentative Test auf die Software-Version 0.1:

Testschema	Testnummer	Schweregrad	Ausfall
Schema 1	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
Schema 2	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	0	
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
	6	10	Es wird keine Priorität bei der Überquerung gesetzt

Tabelle 4. : Ergebnistabelle der repräsentativen Tests auf der Software Version 0.1

Repräsentative Test auf die Software-Version 1.0:

Testschema	Testnummer	Schweregrad	Ausfall
Schema 1	1	0	
	2	7	Zug 2 fährt nicht mehr weiter
	3	7	Zug 2 fährt nicht mehr weiter
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
Schema 2	1	0	
	2	7	Zug 2 fährt nicht mehr weiter
	3	7	Zug 2 fährt nicht mehr weiter
	3	10	Zug 2 hält keinen Abstand zu Zug 3
	3	10	Zug 3 überholt Zug 2
	4	7	Zug 2 fährt nicht mehr weiter
	5	10	Zug 2, bzw. 4 halten keinen Abstand zu den voranfahrenden Zügen
	5	10	Zug 3 überholt Zug 4
	6	10	Es wird kein Abstand zu den voranfahrenden Zügen gehalten
	6	10	Die vorderen Züge werden von hinteren überholt
6	10	Es wird keine Priorität bei der Überquerung gesetzt	

**Tabelle 5. : Ergebnistabelle der repräsentativen Test auf der Software
Version 1.0**

Repräsentative Test auf die Software-Version 2.0 :

Testschema	Testnummer	Schweregrad	Ausfall
Schema 1	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	0	
	3	0	
Schema 2	1	0	
	2	3	Zug 2 hält ohne Grund kurz an
	3	3	Zug 2 hält ohne Grund kurz an
	3	0	
	3	0	
	4	0	
	5	0	
	5	0	
	6	0	
	6	0	
	6	0	

**Tabelle 6. : Ergebnistabelle der repräsentativen Test auf die Software
Version 2.0**

Repräsentative Test auf die Software-Version 3.0 :

Testschema	Testnummer	Schweregrad	Ausfall
Schema 1	1	0	
	2	0	
	3	0	
	3	0	
	3	0	
Schema 2	1	0	
	2	0	
	3	0	
	3	0	
	3	0	
	4	0	
	5	0	
	5	0	
	6	0	
	6	0	
	6	0	

**Tabelle 7. : Ergebnistabelle der repräsentativen Tests auf der Software
Version 3.0**

6. Ergebnisanalyse

6.1. Zuverlässigkeitsanalyse

Um die angewandten Sicherheits- und Zuverlässigkeitsmethoden untereinander vergleichen zu können, müssen wir die Sicherheit und Zuverlässigkeit unserer Software vor der Anwendung der ausgewählten Methoden feststellen.

Berechnen wir zuerst den MTTF Wert für die Software Version 0.1:

Laut Tabelle 4 sind 13 Ausfälle festgestellt worden. Dies ergibt einen MTTF Wert von 138,46 , die Zeit wird in Sekunden gerechnet.

Analog zu den obigen Rechnungen, errechnen wir die Zuverlässigkeit für die drei Sicherheits- und Zuverlässigkeitsmethoden.

Die Zuverlässigkeitswerte werden mit dem ursprünglichen Zuverlässigkeitswert MTTF (initial), nämlich der Zuverlässigkeit der Software Version 0.1, verglichen. Aus diesem Vergleich ergibt sich eine absolute Verbesserung. Die Zuverlässigkeitsverbesserung kann man auch relativ als Prozentsatz ausdrücken.

Methoden	MTTF (initial)	MTTF	Verbesserung (absolute)	Verbesserung (relative)
Inspektion	136,46	128,57	-7,89	-5,78%
Black-Box Tests	136,46	450	313,54	229,77%
Glas-Box Tests	136,46	1801	1664,54	1219,80%

Tabelle 8. : Ergebnisanalyse für die Zuverlässigkeit

Man beobachtet, dass die Zuverlässigkeit nach der Inspektion Methode leicht abgenommen hat, auch wenn diese Methode mehr Ausfälle festgestellt hat als die Glas-Box Methode.

Hier ist eine Auflistung der gefundenen und behobenen Fehler:

- Inspektion: 59 Fehler

- Black-Box Tests: 67 Fehler
- Glas-Box Tests: 40 Fehler

Dies bestätigt, die ursprüngliche Annahme, dass es weniger wichtig ist, wieviele Fehler man mittels einer Methode gefunden und behoben hat, sondern was es für Fehler sind, die behoben werden.

Die Glas-Box Methode hat „nur“ 40 Fehler behoben, jedoch die beste Zuverlässigkeitsverbesserung mit 1219,28% ermöglicht.

Die Black-Box Methode hat nicht so gut abgeschnitten wie die Glas-Box Methode, weil die spezifizierten Tests nicht genau formuliert worden sind, da dem Black-Box Tester weniger Hintergrundwissen zur Verfügung steht als dem Glas-Box Tester. Ein typisches Beispiel dafür ist die Formulierung „Zwei Züge fahren mit geringem Abstand zueinander“ bei den Black-Box Tests, ein ähnlicher Test in den Glas-Box Tests ist aber folgendermaßen formuliert „Zwei Züge fahren mit einem Abstand kleiner als 15 Einheiten“.

Dies führt uns zu der nächsten Analyse, denn nicht nur die Verbesserung der Zuverlässigkeit ist ein entscheidender Faktor für eine Sicherheits- und Zuverlässigkeitsmethode, sondern auch die Effizienz einer Methode.

Die Effizienz richtet sich nach der Zeit, die eine Methode in Anspruch nimmt. Dabei ist nicht nur die Ausführungszeit gemeint, sondern auch die Vorbereitungszeit.

Die absolute Effizienz „ E_a “ wird folgendermaßen errechnet:

Sei V_a die absolute Verbesserung der Zuverlässigkeit nach einer Methode, und T die Zeit, die für diese Methode beansprucht wurde.

$$E_a = V_a / T$$

Und genauso lässt sich die relative Effizienz „ E_r “ errechnen, wenn V_r die relative Verbesserung der Zuverlässigkeit ist, dann ist

$$E_r = V_r / T$$

Die Effizienz ist sozusagen das Qualitätspreisverhältnis bei einer Software. Bei diesem Vergleich wird erst deutlich, wie wichtig eine Aussage über die Effizienz einer Methode ist.

Dabei soll uns folgende Tabelle helfen:

Methode	Zeit (in Minuten)	Effizienz (absolute)	Effizienz (relative)
Inspektion	150	-0,05	-2,31%
Black-Box Tests	80	3,92	172,33%
Glas-Box Tests	150	11,10	487,92%

**Tabelle 9. :Effizienz der Methode bei der
Zuverlässigkeitsmessung**

Aus dieser Prüfung wird es klar, dass in diesem Experiment die Glas-Box Methode nicht nur die Zuverlässigkeit am meisten verbessert hat, sondern diese Methode auch am Effizientesten ist.

6.2. Sicherheitsanalyse

Sowie bei der Zuverlässigkeitsanalyse, muss bei der Sicherheitsanalyse zuerst die Sicherheit vor der Anwendung der Sicherheits- und Zuverlässigkeitsmethoden gemessen werden, nämlich die Sicherheit der Software Version 0.1.

Mittels dem „Expert Opinion“ konnten wir jedem Ausfall einen Schweregrad zuweisen.

Um ein Wert für die Sicherheit zu bekommen, summieren wir die Werte der Spalte Schweregrad in der Tabelle 4, diese Summe nennen wir EO.

Laut Tabelle 4 ist $EO = 102$.

Analog zu den obigen Rechnungen, errechnen wir die Sicherheit für die drei Sicherheits- und Zuverlässigkeitsmethoden.

Die Sicherheitswerte werden mit dem ursprünglichen Sicherheitswert EO (initial), nämlich der Sicherheit der Software Version 0.1, verglichen. Aus diesem Vergleich ergibt sich eine absolute Verbesserung. Die Sicherheitsverbesserung kann man auch relativ als Prozentsatz ausdrücken.

Methode	EO (initial)	EO	Verbesserung (absolute)	Verbesserung (relative)
Inspektion	102	125	-23	-22,55%
Black-Box Tests	102	12	90	88,24%
Glas-Box Tests	102	0	102	100,00%

Tabelle 10. : Ergebnisanalyse für die Sicherheit

Bei dieser Tabelle merkt man deutlich, wie sehr die Auswahl der Methoden die Verbesserung der Softwaresicherheit entscheidet.

Die Inspektion Methode hat am schlechtesten abgeschnitten, denn die Sicherheit hat nach dieser Methode abgenommen. Dies kann man sich ganz einfach mit der Tatsache erklären, dass Änderungen im Code nicht immer den gewünschten Effekt zur Folge haben, vor allem wenn ein Inspektion nicht mit einer weiteren Testmethode

verbunden ist. In diesem Fall wird es für den Entwickler schwierig sein alleine aus der Inspektion Methode eine Verbesserung der Sicherheit zu erreichen.

Die Glas-Box Methode konnte die Sicherheit am meisten verbessern, weil diese Methode systematisch alle möglichen Fälle einkalkuliert und nicht von der Meinung von Menschen (Entwicklern, Testern oder Inspectors) abhängt. Die Glas-Box Methode ist aber nur geringfügig einsetzbar, denn sobald die Software umfangreicher wird, wird es schwer, bzw. unmöglich sein alle Codepfade zu verfolgen und damit alle denkbaren Fälle zu decken.

Die Black-Box Methode hat zwar auch die Sicherheit verbessert, jedoch nur gering im Vergleich mit der Glas-Box Methode. Das liegt daran, dass diese Methode sehr stark von menschlichen Faktoren abhängig ist, denn die Spezifikation wurde von einem Menschen verfasst und das Auswählen der Test-Suite aus dieser Spezifikation ist auch nicht sehr von dem Tester abhängig.

Ein letzter Faktor ist die Effizienz der Sicherheits- und Zuverlässigkeitsmethoden im Hinblick auf die Sicherheit.

Werden die Effizienzwerte, wie im letzten Abschnitt beschrieben, berechnet, dann ergibt sich folgende Tabelle:

Methode	Zeit (in Minuten)	Effizienz (absolute)	Effizienz (relative)
Inspektion	150	-0,15	-9,02%
Black-Box Tests	80	1,125	66,18%
Glas-Box Tests	150	0,68	40,00%

Tabelle 11. : Effizienz der Methode bei der Zuverlässigkeitsmessung

Noch einmal beweist sich die Inspektion für unser Experiment als unpassend, denn auch bei der Effizienz ist diese Methode sehr schlecht.

Die Glas-Box Methode hat einen guten Effizienzwert erreicht, ist aber nicht die effizienteste Methode.

Die Black-Box Methode hat zwar die Sicherheit nicht am meisten verbessert, hat aber in Punkto Effizienz das beste Ergebnis erreicht.

7. Abschluss

Am Ende unseres Experimentes kann man von einem eindeutigen Ergebnis sprechen, denn die Zahlen haben die Glas-Box Methode als eindeutigen Sieger diesen Vergleiches ausgezeichnet.

Die Black-Box Methode ist ebenfalls empfehlenswert, denn sie hat bei der Effizienz bezüglich der Sicherheit ein gutes Ergebnis erreicht.

Es ist auffällig, dass die Verbesserungskurven der Sicherheit und der Zuverlässigkeit parallel verlaufen, und zwar bei allen drei Methoden. Es ist klar, dass eine Verbesserung der Sicherheit auch eine Verbesserung der Zuverlässigkeit nach sich zieht, doch hier verläuft die Verbesserung der Zuverlässigkeit äußerst proportional zur Verbesserung der Sicherheit.

Ich glaube, man kann allgemein sagen, dass die Glas-Box Methode für kleine bis mittlere Software, deren Quellcode nicht 10.000 Zeile überschreitet, am besten geeignet ist. Der Grund ist, dass in dieser Größenordnung noch alle Code-Pfade verfolgbar sind.

Die Black-Box Methode hat ebenfalls wesentlich zur Verbesserung der Sicherheit und Zuverlässigkeit beigetragen, nur könnte sie erst dann eine bessere Alternative zur Glas-Box Methode werden, wenn die Software umfangreicher ist, da diese Methode sich weniger nach dem Code orientiert, sondern vielmehr nach der Softwarespezifikation.

Am schlechtesten ist bei unserem Vergleich die Inspektion Methode ausgefallen. Es könnte daran liegen, dass Inspektionen ausschließlich von Experten, die sich in diesem Softwarebereich spezialisiert haben, durchgeführt werden müssen. Aus diesem Grund empfiehlt sich diese Methode nicht für Entwickler, die in verschiedenen Softwarebereichen tätig sind, da sie als Inspector unbedingt über eine ausgeprägte Erfahrung und ein umfangreiches Know-how in ein bestimmten Gebiet verfügen sollten.

Die Inspektion Methode empfiehlt sich daher für größere Firmen, die eine eigene Test-Abteilung besitzen, die ausschließlich Inspektionen im selben Softwaregebiet durchführt . In einem solchen Fall könnte die Inspektion bessere Ergebnisse, was die Verbesserung der Sicherheit und Zuverlässigkeit angeht, erzielen.

Die Inspektion hat aber andere Merkmale der Softwarequalität verbessert, wie Änderbarkeit, denn der Code ist lesbarer geworden, und somit leichter zu pflegen.

Es gibt vieles was man noch tun könnte um die Ergebnisse und Analysen auszubauen, die aus dieser Arbeit resultierten.

Man könnte untersuchen, wie umfangreich eine Software höchstens sein darf, damit die Glas-Box Methode noch die besten Ergebnisse erzielt, was die Verbesserung der Sicherheit und Zuverlässigkeit sowie die Effizienz betrifft.

Man könnte auch untersuchen, ob die Inspektion Sinn macht, wenn sie in Kombination mit einer weiteren Methode, z.B. der Black-Box Methode angewandt wird. Und zwar auch für Entwickler die keine erfahrene Experten sind. Denn es könnte sein, dass die zwei Methoden sich kompensieren, indem die Inspektion die Mängel der Black-Box Methode behebt, die dadurch entstehen, dass der Tester weniger Hintergrundinformationen über den Quellcode besitzt. Genauso wie die Black-Box Methode die Mängel ausgleichen könnte, die dadurch entstehen, dass der Benutzer nicht unbedingt ein erfahrener Experte in diesem Bereich ist, dadurch, dass der Tester immer die Folgen seiner Änderung im Programm vor Auge behält.

Es ist schwierig, genauer, fast unmöglich eine allgemeine Aussage zu geben, welche Methode grundsätzlich besser ist. Man kann als Abschluss dieser Arbeit sagen, dass die Auswahl der richtigen Sicherheits- und Zuverlässigkeitsmethode davon abhängt, **was** für Software entwickelt werden muss und **wer** die Software entwickeln soll.

Der größte Mangel des Experiments ist, dass die Ergebnisse, statistisch gesehen, nicht sehr aussagekräftig sind, da die Werte nur einmalig ausgerechnet worden sind. Ein weiterer Mangel bei der Inspektion ist, dass die Inspektoren keine Experten waren.

Bibliographie / Links

- [1] John D. Musa, Anthony Iannino, and Kazuhira Okumoto (1987).
Software Reliability: Measurement, Prediction, Application. McGraw Hill College
- [2] R. Lyu (1996). Handbook of Software Reliability Engineering. McGraw Hill Text
- [3] Timm Grams, Fachhochschule Fulda (2004).
Grundlagen des Qualitäts- und Risikomanagements
- [4] Torsten Licht, RWTH Aachen (2004).
Zuverlässigkeit von Systemen und Ihre Messbarkeit
- [5] Stefan Wagner, TU München (2004). Experiments about the Effectiveness of
Reliability and Safety Methods.
- [6] Stefan Wagner, TU München (2004). Reliability efficiency of defect-detection
techniques: A field study.
- [7] Lars Jansen (1997). Referenzfallstudie Verkehrsleittechnik.
- [8] Arno Gramatke (2003). Qualitätssicherung für die Softwareentwicklung.
- [9] Prof. Dr. Klaus Peter Fähnrich (2002). Kernfachvorlesung Praktische
Informatik, Sommersemester 2002.
- [10] M. Fagan (1976). Design and Code Inspections to Reduce Errors in Program
Development. IBM Systems Journal 15, 3 (1976).
- [11] G. J. Myers (1991). Methodisches Testen von Programmen. R. Oldenbourg,
München, Wien.
- [12] Christopher Fox (1999). Java Inspection Checklist.
http://www.cs.toronto.edu/~sme/CSC444F/handouts/java_checklist.pdf