Software ist heute überall. Ob in Smartphones, Autos oder Bankautomaten, überall sorgt sie dafür, dass Dinge reibungslos ablaufen. Deshalb ist es umso wichtiger, dass diese Software zuverlässig funktioniert. Fehler können nicht nur nervig sein, sondern in sicherheitskritischen Bereichen auch gefährlich werden.

Normalerweise versucht man solche Fehler mit Testing zu finden, indem man das Programm mit ausgewählten Eingaben ausführt und überprüft, ob alles wie erwartet funktioniert. Doch Tests haben ihre Grenzen: Sie prüfen nur bestimmte Szenarien und decken nie alle möglichen Eingaben ab. Ein Programm kann also alle Tests bestehen und trotzdem in anderen Fällen noch fehlerhaft sein. Gerade bei komplexen oder sicherheitskritischen Systemen reicht Testing deshalb oft nicht aus, um wirklich Vertrauen in die Software zu haben. Hier kommen formale Methoden ins Spiel. Sie helfen uns, Programme mathematisch zu beweisen, so wie man auch in der Mathematik einen Satz beweist. Statt nur ein paar Beispiele zu testen, wird sichergestellt, dass das Programm immer richtig funktioniert. Egal, was für Eingaben man wählt.

Das klingt erstmal kompliziert, aber moderne Tools wie Dafny machen das Ganze viel zugänglicher. Dafny ist eine Sprache, die es ermöglicht, Programme direkt mit Bedingungen wie Vor- und Nachbedingungen, Schleifeninvarianten und Terminationsprüfungen zu versehen. Damit kann man schon beim Schreiben prüfen, ob das Programm korrekt ist, ohne es überhaupt auszuführen. So lassen sich viele Fehler schon früh erkennen und beheben, bevor sie später große Probleme verursachen. Gerade in sicherheitskritischen Systemen kann das einen großen Unterschied machen.

Dass Fehler in Software nicht nur theoretisch vorkommen, zeigen leider immer wieder reale Beispiele. Vor einiger Zeit musste Tesla rund zwei Millionen Fahrzeuge zurückrufen, weil es beim Autopiloten zu Softwarefehlern gekommen war[1](#user-content-fn-1). Das System hat bestimmte Hindernisse nicht korrekt erkannt, was im schlimmsten Fall zu Unfällen hätte führen können. Ein anderes Beispiel ist die NASA. Beim Mars Rover Spirit führte ein Speicherleck dazu, dass das System immer wieder abstürzte[2](#user-content-fn-2). Das Problem war schwer zu finden und hätte die Mission beinahe zum Scheitern gebracht.

Solche Fälle zeigen, wie wichtig es ist, Programme nicht nur zu testen, sondern ihre Korrektheit von Anfang an mathematisch zu überprüfen. Ein Test hätte die Tesla-Fehler vielleicht nicht gefunden, weil er nur einen Bruchteil aller möglichen Szenarien abdeckt. Mit formalen Methoden kann man hingegen sicherstellen, dass das Programm in allen denkbaren Fällen korrekt funktioniert. Das gibt nicht nur den Entwicklern mehr Sicherheit, sondern vor allem auch den Menschen, die sich auf diese Systeme verlassen müssen.

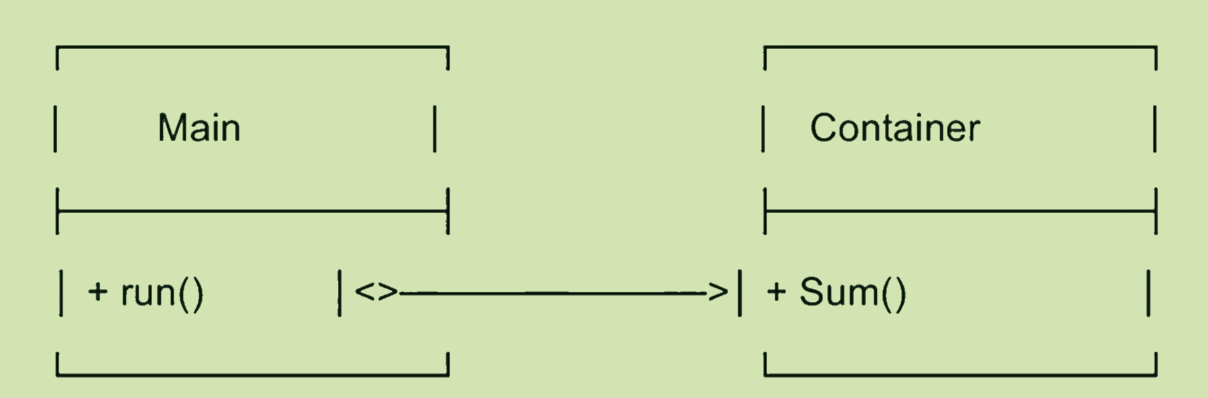
**Problem**

Die Entwicklung korrekter Software ist in der Praxis eine der größten Herausforderungen überhaupt. Schon beim Schreiben von Code müssen Entwickler darauf achten, dass keine Fehler entstehen, die sich später nur schwer finden lassen. Besonders in sicherheitskritischen Systemen, in denen Software über Leben und Tod entscheiden kann, sind Fehler nicht nur ärgerlich, sondern können gravierende Folgen haben.

Während Tests und Code-Reviews helfen können, Fehler zu erkennen, reichen diese Methoden oft nicht aus, um die Korrektheit eines Programms in allen denkbaren Situationen sicherzustellen. Das liegt daran, dass Tests immer nur eine Auswahl an Eingaben prüfen können, niemals aber alle Möglichkeiten abdecken. Ein Programm kann also alle Tests bestehen und trotzdem noch fehlerhaft sein.

Genau hier setzt die formale Verifikation an. Das Ziel ist es, Programme bereits beim Schreiben so zu entwickeln, dass ihre Korrektheit mathematisch bewiesen werden kann. Damit lässt sich garantieren, dass Programme für alle denkbaren Eingaben korrekt arbeiten.

An diesem Punkt kommt Dafny ins Spiel. Dafny ist ein modernes Tool, das speziell für die Verifikation von Programmen entwickelt wurde. Es erlaubt es Entwicklern, ihre Programme nicht nur zu schreiben, sondern gleichzeitig auch deren Korrektheit zu beweisen. Das bedeutet, dass die Software noch vor der Ausführung auf mögliche Fehler überprüft wird. Das erleichtert die Entwicklung von zuverlässiger Software enorm und sorgt dafür, dass man sich später nicht aufwendig um schwer nachvollziehbare Bugs kümmern muss.

In dieser Arbeit wird deshalb untersucht, wie Dafny konkret dabei hilft, die Korrektheit von Programmen bereits beim Schreiben zu garantieren. Um die Funktionsweise des Beispielprogramms (Unter DAFY/code ) zu verdeutlichen, wird in der folgenden Abbildung ein Klassendiagramm dargestellt:

**Beschreibung:**

Main ruft Container.Sum() auf.

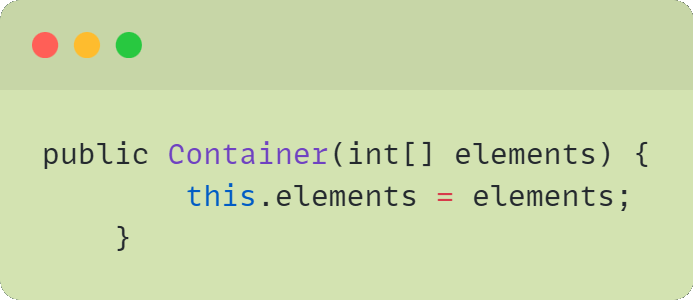
* Container ist eine Klasse, welches eine Methode besitzt.
* Sum() ist eine Methode, die Sequenze addiert.

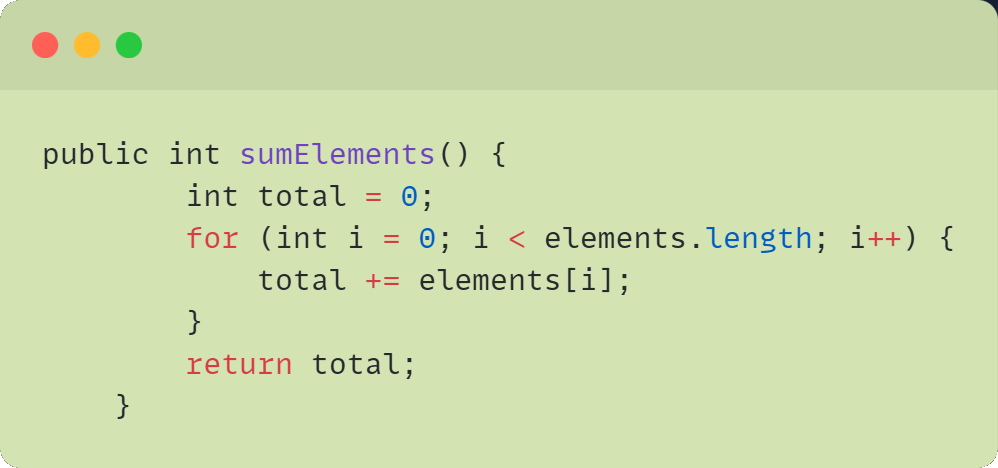
**Dafny Features:**

Dafny erweitert die Programmentwicklung mit praktischen Funktionen, die die Korrektheit von Programmen bereits beim Schreiben sicherstellen. Diese Features machen es einfacher, Programme mathematisch zu beweisen und potenzielle Fehler frühzeitig zu erkennen:

* Pre- und Postconditions:  
  Sie legen Bedingungen fest, die vor und nach der Ausführung einer Funktion erfüllt sein müssen. So wird präzise definiert, was eine Funktion leisten darf und was sie garantiert zurückliefert (1).
* Loop Invariants:  
  Sie beschreiben eine Bedingung, die zu Beginn und nach jedem Schleifendurchlauf erfüllt sein muss. Damit kann man mathematisch beweisen, dass die Schleife zu jedem Zeitpunkt korrekt arbeitet (2).
* Termination Checking:  
  Dieses Feature prüft, ob das Programm oder eine Schleife garantiert irgendwann endet und nicht in einer Endlosschleife hängen bleibt (3).
* SMT-Solver (Z3):  
  Ein leistungsstarkes Tool, das die Einhaltung aller spezifizierten Bedingungen automatisch überprüft und so die Verifikation des Codes unterstützt (4).

Beispiele:

Z1:

Z2:

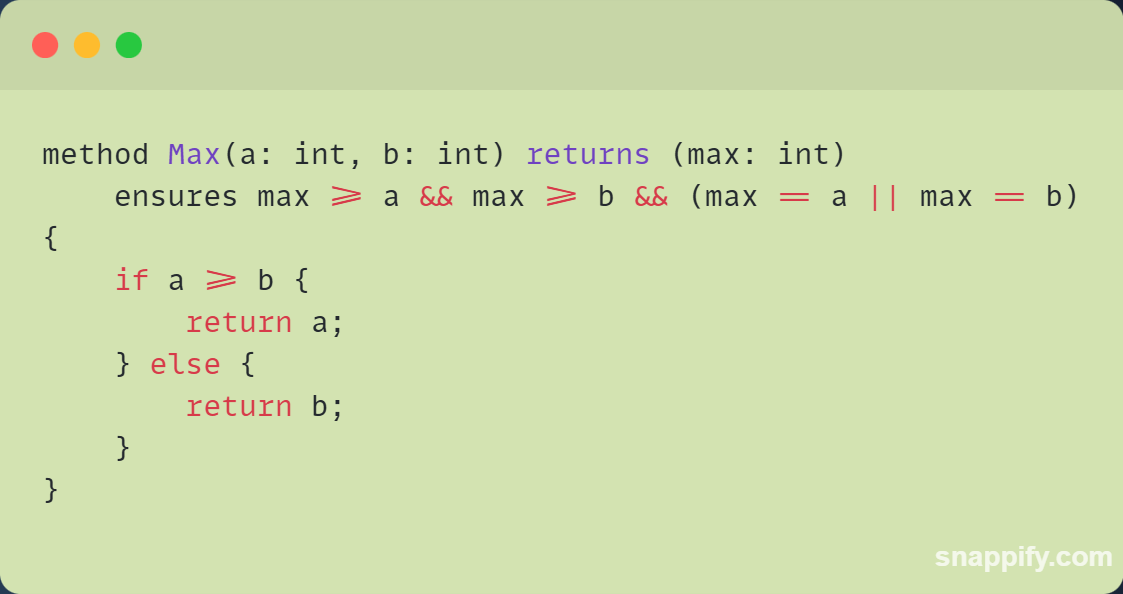
Z3:

Main Idea & Details:

Um die praktische Anwendung von Dafny besser zu verstehen, werden im Folgenden die zentralen Sprachkonstrukte anhand eines einfachen Beispiels gezeigt. Besonders wichtig sind dabei die Möglichkeiten, mathematische Bedingungen direkt im Code zu formulieren und damit die Korrektheit automatisch überprüfen zu lassen.

**Syntax und Sprachkonstrukte (Code als Beispiel)**

Dafny orientiert sich an bekannten Syntaxmustern moderner Programmiersprachen und ergänzt sie durch spezielle Schlüsselwörter für Spezifikationen. So lässt sich zum Beispiel eine Funktion schreiben, die das Maximum zweier Zahlen berechnet:



Dieses Beispiel zeigt, wie sich eine Nachbedingung (ensures) direkt im Code definieren lässt, sodass die Verifikation bereits beim Schreiben erfolgt.

**requires und ensures**

requires beschreibt Vorbedingungen, die vor dem Aufruf einer Funktion erfüllt sein müssen. ensures definiert die Bedingungen, die nach Ausführung der Funktion gelten sollen. Mit diesen beiden Schlüsselwörtern lässt sich das gewünschte Verhalten einer Funktion präzise festlegen und überprüfen.

**assert und invariant**

Das Schlüsselwort assert wird genutzt, um an beliebigen Stellen im Code Bedingungen zu formulieren, die immer gelten müssen. Wenn eine Bedingung nicht erfüllt wird, meldet der Verifier einen Fehler. Mit invariant lassen sich Schleifenbedingungen festlegen, die vor, während und nach jeder Iteration erfüllt sein müssen. Dadurch wird sichergestellt, dass Schleifen keine unerwünschten Zustände erreichen.

**Z3-Integration**

Dafny verwendet im Hintergrund den SMT-Solver Z3, um die definierten Bedingungen automatisch zu überprüfen. Dadurch werden auch komplexe Bedingungen schnell und zuverlässig geprüft, ohne dass Entwickler jeden Beweis manuell führen müssen.

**Eigene Erfahrung**

Bislang konnte ich noch keine eigenen Erfahrungen mit Dafny sammeln, daher kann ich derzeit noch keine Hürden oder Herausforderungen berichten. Dieser Abschnitt dient als Platzhalter und wird in der nächsten Abgabe ergänzt, sobald ich mit Dafny in der Praxis gearbeitet habe.

**Literature Critique**

In dieser Literaturkritik wird das Paper **„An Automatic Program Verifier for Functional Correctness“** von **K. Rustan M. Leino (2010, Microsoft Research)** betrachtet. Viele unserer Aussagen in dieser Arbeit bauen auf den Ergebnissen und Erkenntnissen dieses Papers auf, da es Dafny sehr umfassend beschreibt und anhand von Fallstudien erklärt. Obwohl es weitere Papers gibt, erschien uns dieses am passendsten, da es eine gute Mischung aus Praxis und Theorie bietet und viele grundlegende Dafny-Features detailliert behandelt.

Literaturkritik ist wichtig, um die Stärken und Schwächen eines Papers zu verstehen und einzuschätzen, inwiefern es für eigene Arbeiten und Anwendungen relevant ist. Dabei helfen strukturierte Einschätzungen, gezielt die eigenen Forschungslücken zu erkennen und einen objektiven Überblick über die verwendete Literatur zu geben.

**Besonders gut verständlich war:**

* **Überblick über Dafny-Features:** Das Paper erklärt zentrale Konzepte wie Pre-/Postconditions, Loop Invariants und Termination Checking auf eine zugängliche Weise, sodass auch Einsteiger einen guten Überblick gewinnen.
* **Integration des SMT-Solvers Z3:** Die Funktionsweise des SMT-Solvers wird praxisnah dargestellt und erleichtert das Verständnis, wie Dafny mathematische Bedingungen automatisch überprüft.
* **Fallstudien mit Code und Metriken:** Die detaillierte Analyse der Fallstudien, inklusive Lines of Code und Proof Obligations, veranschaulicht die praktische Nutzung von Dafny sehr anschaulich.

**Schwierig zugänglich war:**

* **Dynamic Frames:** Dieses Konzept wurde nur knapp angesprochen und für Einsteiger wenig greifbar erklärt.
* **Proof-Strategien:** Die Darstellung der Lemmas und Invarianten setzte teilweise ein höheres Niveau an formaler Logik voraus, was für weniger erfahrene Leser (wie wir) eine Hürde darstellen kann.

Bisher haben wir online keine bessere Quelle gefunden, die sowohl Einsteiger als auch Fortgeschrittene so gut abholt. Ein möglicher Grund dafür ist, dass andere Papers oft entweder zu theoretisch oder zu speziell sind, während dieses Paper einen guten Überblick bietet und dabei praxisnahe Beispiele liefert. Außerdem ist Dafny erst seit etwa 2010 weltweit intensiver im Einsatz, weshalb es noch nicht so viele leicht verständliche Papers dazu gibt.

**Future Work & Conclusion**

**Fazit**  
In dieser Arbeit wurde gezeigt, dass Dafny als modernes Verifikationstool eine wertvolle Ergänzung zu klassischen Tests sein kann. Gerade in sicherheitskritischen Systemen, bei denen Fehler schwerwiegende Folgen haben, ermöglicht Dafny eine frühzeitige Fehlererkennung bereits während des Schreibens. Dafny ist nicht nur für kleine Projekte interessant, sondern bietet auch für Studierende einen praxisnahen Einstieg in die formale Verifikation.

**Ausblick**  
Für die Zukunft sollte der Fokus darauf liegen, Dafny noch stärker in größere Teams und agile Entwicklungsprozesse zu integrieren. Gerade in Projekten, in denen viele Entwickler parallel arbeiten, wird die Koordination von Code-Reviews und Tests immer komplexer. Hier kann Dafny mit seinen formalen Spezifikationen eine verlässliche Grundlage schaffen, die bisher oft fehlt. Derzeit ist der Einsatz in großen Teams noch eine Herausforderung, da viele Unternehmen auf schnelle Ergebnisse setzen und ungern zusätzliche Ressourcen für formale Verifikation einplanen. Dabei wird häufig unterschätzt, dass die Kosten für Fehler im laufenden Betrieb oder bei sicherheitskritischen Anwendungen später deutlich höher sind als die Investition in eine gründliche Verifikation.

Besonders relevant wird Dafny in einer Zeit, in der Künstliche Intelligenz längst ein fester Bestandteil unseres Alltags ist. Systeme mit KI entscheiden heute über Prozesse, die große Auswirkungen haben können. Gerade deshalb ist es wichtig, dass der Code fehlerfrei arbeitet und die Ergebnisse zuverlässig sind. Hier kann Dafny helfen, die Korrektheit solcher Systeme zu gewährleisten – ob in Industrieprojekten, bei kritischen Systemen oder gar bei Software für Weltraummissionen.

Gleichzeitig sollte man bedenken, dass Dafny auch Ressourcen verbraucht und nicht für jedes Projekt zwingend notwendig ist. Nicht jede Software ist sicherheitskritisch. Es bleibt also jedem Team selbst überlassen, zu bewerten, ab wann ein Projekt so kritisch ist, dass keine Fehler erlaubt sein dürfen. Genau dann sollte Dafny eingesetzt werden.

Eine spannende Perspektive wäre es auch, Dafny künftig noch enger in moderne DevOps-Umgebungen zu integrieren, zum Beispiel mit Docker, um Fehler in Datenbanken oder komplexen Systemen zu vermeiden.

Abschließend bleibt es eine Herausforderung, Unternehmen, große Teams und die Community davon zu überzeugen, dass der anfängliche Mehraufwand eine Investition in Qualität und Sicherheit ist. Gerade heute, wo viele Unternehmen auf schnelle Gewinne und Kostenreduzierung setzen, wird oft übersehen, dass ein späterer Fehler teurer sein kann als eine solide Verifikation. Hier kann Dafny eine Brücke zwischen modernen Entwicklungsprozessen und sicherer Software schlagen – eine Aufgabe, die die Community gemeinsam vorantreiben sollte.

1 Vgl. Die Presse, „Probleme mit Autopilot: Tesla ruft zwei Millionen Autos zurück“, 2025.

2 Vgl. NASA Mars Exploration Rover Spirit – Technical Report (NASA JPL), 2004.