Das Expression Problem: Lösung mit Generics in TS[*†]

Mohammadreza Javadikouchaksaraei

Technische Universität Dortmund
Informatik Student mit Matrikelnummer 234019
mohammadreza.javadikouchaksaraei@tu-dortmund.de

Inhaltsverzeichnis

[*]TS: TypeScript

[†]Der vorliegende Titel basiert auf Mads Torgersens Papier „The Expression Problem Revisited“. Es behandelt Definitionen und Konzepte zur Lösung und Implementierung in TypeScript für den Proseminar-Kurs.

1 Einleitung

Das „Expression Problem“ stellt seit jeher eine Herausforderung im Design von Programmiersprachen dar, die die Gestaltung und Implementierung objektorientierter und funktionaler Sprachen maßgeblich beeinflusst. Das Ziel besteht in der Definition eines Datentyps, welcher die Hinzufügung neuer Fälle zum Datentyp sowie die Implementierung neuer Funktionen über den Datentyp ohne erneute Kompilierung des Codes ermöglicht, während gleichzeitig die statische Typensicherheit bewahrt wird [9]. Es gibt verschiedene Ansätze zur Lösung dieses Problems, die jeweils ihre eigenen Vor- und Nachteile haben.

In dieser Ausarbeitung für den Proseminarkurs wurden klassische und moderne Methoden zur Lösung des *Expression Problems* mit *Generics* analysiert, basierend auf dem Artikel „The Expression Problem Revisited: Four new solutions using generics“ [8]. Das Ziel ist es, dieses Ansatz in TypeScript zu implementieren und zu testen.

2 Grundlagen des Expression Problems

2.1 Beschreibung des Expression Problems

2.1.1 Historie

Um die Genese des vorliegenden Problems zu verstehen, ist ein Blick auf die Arbeit von John Reynolds aus dem Jahr 1975 erforderlich. Reynolds identifizierte das Problem in seinen Untersuchungen zur Datenverwaltung in der Programmierung und präsentierte zwei Ansätze: benutzerdefinierte Typen (ADTs[1]) und prozedurale Datenstrukturen [5]. Im Jahr 1990 erweiterte William Cook die Ideen Reynolds' und führte eine Matrix der Darstellungen und Verhaltensweisen in der Datenabstraktion ein. Dabei betonte er die Bedeutung der statischen Typisierung und demonstrierte, wie sich ADTs und Objekte ergänzen können [1].

Im Jahr 1998 benannte Philip Wadler dieses als *the Expression Problem* und präsentierte es als klar definierte Herausforderung, nachdem er mit dem Programmierungsteam der Rice University diskutiert hatte. Er argumentierte, dass die Fähigkeit einer Sprache, dieses Problem zu lösen, ein Indikator für ihre Ausdruckskraft sei.
Wadlers Beitrag hat die wissenschaftliche Gemeinschaft auf die Problematik des Designs von Programmiersprachen aufmerksam gemacht und ist bis heute von Relevanz [9].

2.1.2 Definition und Beispiel

Das Expression Problem Konzept sucht nach Lösungen, um den Datentyp zu erweitern und neue Funktionen hinzuzufügen, ohne den vorhandenen Code neu kompilieren zu müssen [9].

In objektorientierten Sprachen werden Daten als „Klassen“ und Funktionen als „Methoden“ bezeichnet. Die Erweiterung einer Klasse um Unterklassen ist ein relativ einfacher Prozess, wohingegen das Hinzufügen einer neuen Methode eine Änderung der Haupt- und Unterklasse und damit eine Neukompilierung erfordert [2].

[1] ADTs: Abstrakte Datentypen, eine Methode, bei der alle Informationen zur Datenrepräsentation zentral zusammengefasst werden.

In funktionalen Sprachen werden Daten durch Konstruktoren und Funktionen durch Fallunterscheidungen definiert. Neue Funktionen können ohne Änderung des Datentyps hinzugefügt werden. Das Hinzufügen eines neuen Konstruktors erfordert jedoch eine Erweiterung aller Fallunterscheidungen, was auch eine Neukompilierung erforderlich macht [3].

Beispiel 1: Ein Hauptbeispiel für das Verständnis dieses Konzepts

Stellen Sie sich eine Social-Media App wie X (früher Twitter) vor, die zunächst nur das Posten von Text unterstützt. Im Laufe der Zeit wird die Funktionalität erweitert, um auch Bilder, Videos und andere Medien zu ermöglichen, die Textbeiträge erweitern. Schließlich soll eine neue Funktion hinzugefügt werden, die das Teilen von Posts erlaubt (ein Shareable-Post). In einem idealen Modell sollten Entwickler solche Funktionen hinzufügen können, ohne die bestehenden Datenstrukturen zu ändern oder den gesamten Code neu kompilieren zu müssen.

Dieses Beispiel 1 illustriert das Kernproblem des Expression Problems: die Erweiterbarkeit von Software, ohne bestehenden Code zu modifizieren.

Die Lösungen für das Expression Problem ermöglichen die einfache Entwicklung und Erweiterung von Anwendungen durch spezifische Paradigmen und Designmuster [9]. Diese Ansätze helfen Entwicklern, robuste und flexible Programme zu erstellen, die schnell auf Benutzerbedürfnisse reagieren und mit neuen Technologien kompatibel sind.

2.2 Klassische Ansätze

2.2.1 Klassen-basiert

Ein klassenbasierter Ansatz zur Lösung des Expression Problems nutzt die objektorientierte Programmierung, bei der neue Funktionen durch Erweiterung (Subklassifizierung) und Vererbung bestehender Klassen implementiert werden. Diese Methode ermöglicht es, dass Subklassen Eigenschaften ihrer Basisklassen erben und modifizieren, wodurch die Erweiterbarkeit des Codes ohne Neukompilierung gewährleistet wird [2].

Im Beispiel 1 der Social-Media-App sollen neue Funktionen und Datentypen integriert werden. Dafür wird das Interface `IPost` erstellt, welches die grundlegenden Funktionen für alle Post-Typen definiert. Auf dieser Basis können spezialisierte Datentypen wie `TextPost`, `ImagePost` und weitere entwickelt werden. Wenn eine neue Funktion hinzugefügt werden soll..

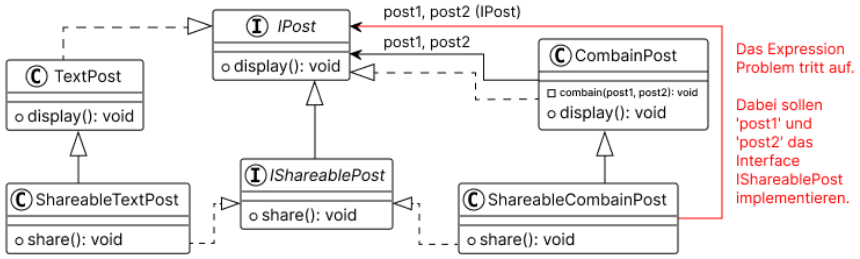

Abbildung 1: Klassen-basiert UML-Diagramm

Wie in Abbildung 1 gezeigt, fügt die `share()`-Funktion hinzu, indem sie `IPost` erweitert. Probleme entstehen, wenn zusätzliche Funktionen wie `CombinePost` integriert werden sollen, da `post1` und `post2` ebenfalls das Interface `IShareablePost` implementieren müssen. Dies verdeutlicht das Expression Problem und erfordert Anpassungen am Code.

Vorteile und Nachteile des klassenbasierten Ansatzes

Der klassenbasierte Ansatz zur Lösung des *Expression Problems* ist leicht verständlich und fördert die Code-Wiederverwendung durch Vererbung [2]. Entwickler können bestehende Funktionalitäten einfach erweitern und wiederverwenden [1], was die Entwicklung und Wartung erleichtert [8].

Nachteile sind, dass Erweiterungen Änderungen in der Basisklasse erfordern und zu einer Neukompilierung führen [9]. Neue Funktionen können den bestehenden Code beeinträchtigen und Anpassungen in allen abgeleiteten Klassen notwendig machen [5], was die Wartbarkeit erschwert und das Fehlerrisiko erhöht [3].

2.2.2 Visitor-Pattern

Das Visitor-Pattern ist ein Verhaltensmuster in der objektorientierten Programmierung, das die Trennung von Algorithmen und den Objektdatenstrukturen ermöglicht [2]. Es erlaubt das Hinzufügen neuer Operationen, indem eine Visitor-Klasse eingeführt wird, die die zu implementierenden Methoden für verschiedene Objekttypen definiert. Dieses Muster bewahrt die Integrität der ursprünglichen Klassen, indem es die Notwendigkeit vermeidet, deren Code zur Erweiterung von Funktionalitäten zu modifizieren [2].

Im Beispiel 1 der Social-Media App könnte das Visitor-Pattern verwendet werden, um unterschiedliche Funktionalitäten wie *Display* und *Shareable* zu implementieren, ohne die Post-Klassen selbst zu ändern. Dies wird durch die Einführung einer Visitor-Klasse erreicht, die spezifische Methoden für jede Art von Nachricht bereitstellt, sodass neue Operationen hinzugefügt werden können, ohne den bestehenden Code anzupassen.

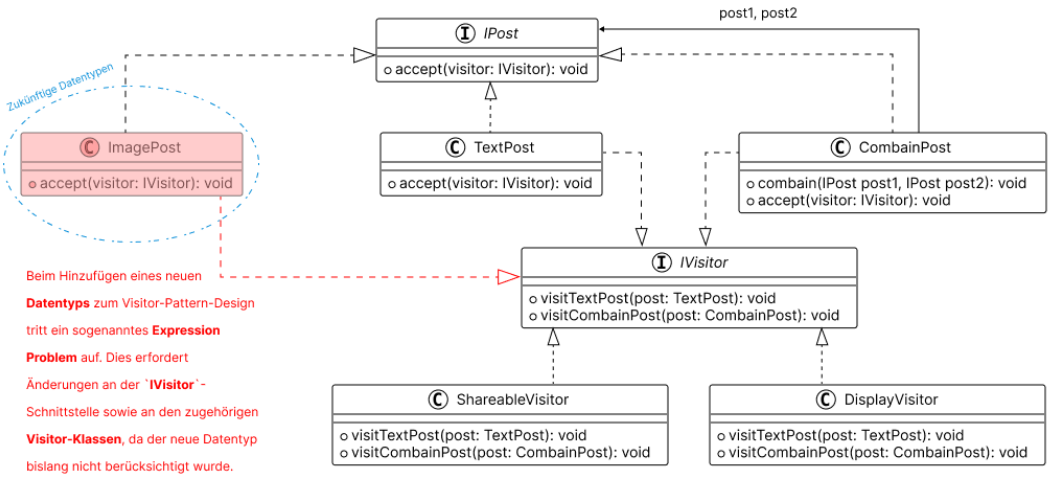

Abbildung 2: Visitor-Pattern UML-Diagramm

Im UML-Diagramm Abbildung 2 wird gezeigt, wie die Klassen `DisplayVisitor` und `ShareableVisitor` verschiedene Operationen auf `TextPost` und `CombinPost` anwenden. Das Visitor-Pattern ermöglicht neue Operationen, ohne die Post-Klassen zu ändern. Bei der Hinzufügung neuer Datentypen müssen jedoch auch die Visitor-Klassen aktualisiert werden, was das Expression Problem verdeutlicht und Codeanpassungen erfordert.

Vorteile und Nachteile des Visitor-Patterns

Das Visitor-Pattern ermöglicht es, neue Operationen hinzuzufügen, ohne die bestehenden Klassen zu ändern, was die Flexibilität und Erweiterbarkeit des Codes erhöht [2]. Eine klare Trennung der Operationen von der Datenstruktur trägt dazu bei, den Code besser strukturiert und wartbarer zu halten [1]. Dieses Designmuster bietet somit eine effiziente Möglichkeit, die Funktionalität eines Systems zu erweitern, ohne tiefgreifende Änderungen an den bestehenden Klassen vorzunehmen [8].

Jedoch kann das Visitor-Pattern zu komplexem und schwer wartbarem Code führen, insbesondere bei großen Objektstrukturen [9]. Jede neue Klasse erfordert das Hinzufügen entsprechender Besuchermethoden, was den Entwicklungsaufwand erhöhen kann und die Übersichtlichkeit des Codes beeinträchtigt [5]. Trotz dieser Herausforderungen bleibt das Visitor-Pattern ein wertvolles Werkzeug zur Strukturierung von komplexen Systemen und zur Trennung von Daten und Operationen [3].

3 Generics und Programmiersprachenwahl

3.1 Einführung und Zweck von Generics

Generics stellen ein grundlegendes Konzept der modernen Programmierung dar, welches die Definition von Typen als Parameter ermöglicht. Dadurch können Funktionen und Klassen so gestaltet werden, dass sie mit verschiedenen Datentypen arbeiten, ohne dass diese Datentypen explizit für jede Instanz spezifiziert werden müssen. Dies führt zu einer höheren Flexibilität und Wiederverwendbarkeit des Codes, da Algorithmen und Datenstrukturen unabhängig von konkreten Datentypen entwickelt werden können [8].

Die Anwendung von Generics zielt darauf ab, die Typensicherheit und Lesbarkeit des Codes zu erhöhen. Die Kompilierungszeit-Überprüfung der Typen minimiert die Notwendigkeit von Typumwandlungen, wodurch die Wahrscheinlichkeit von Laufzeitfehlern reduziert wird und die Wartbarkeit des Codes verbessert wird [3].

Generics bieten zahlreiche Vorteile, wie:

- **Typensicherheit**: Generics ermöglichen die Überprüfung der Typen zur Kompilierungszeit, was dazu beiträgt, Laufzeitfehler zu minimieren und die Zuverlässigkeit des Codes erheblich zu steigern [9].
- **Wiederverwendbarkeit**: Algorithmen und Datenstrukturen, die mit Generics geschrieben wurden, können mit verschiedenen Datentypen verwendet werden, ohne dass sie mehrfach implementiert werden müssen. Dies spart Zeit und Aufwand bei der Entwicklung [8].
- **Lesbarkeit und Wartbarkeit**: Generics reduzieren die Notwendigkeit von Typumwandlungen und machen den Code dadurch einfacher zu lesen und zu verstehen. Dies fördert die Wartbarkeit und ermöglicht eine schnellere Fehlerbehebung [9].

3.2 Vorstellung der gewählten Programmiersprache

Die Wahl von TypeScript als Programmiersprache für die Implementierung der Lösungsansätze basiert auf meiner persönlichen Erfahrung sowie den umfassenden Möglichkeiten, die TypeScript für das Expression Problem bietet.

Was ist TypeScript? TypeScript ist eine von Microsoft entwickelte Open-Source-Programmiersprache, die als typisierte Obermenge von JavaScript fungiert und in reines JavaScript transpiliert. Die Sprache entstand aus ECMAScript, insbesondere nach bedeutenden Änderungen in der JavaScript-Programmierung durch die V8-Engine von Google. TypeScript bietet eine strenge Typisierung und unterstützt sowohl objektorientierte als auch imperative Programmierkonzepte. Es erweitert JavaScript um optionale statische Typen, Klassen, Module und Interfaces und stellt somit eine leistungsstarke Werkzeugkette für die Entwicklung großer JavaScript-Anwendungen dar [7].

Gründe für die Wahl von TypeScript: Es wurde aufgrund seiner starken Unterstützung für Generics ausgewählt, was die flexible und typsichere Implementierung von Lösungen für das Expression Problem ermöglicht [8]. Die statische Typisierung von TypeScript hilft, Fehler frühzeitig zu erkennen und sicherzustellen, dass Erweiterungen von Datentypen und Operationen korrekt typisiert sind [6]. Zudem profitiert TypeScript von einer großen Entwicklergemeinschaft und einer Vielzahl an Bibliotheken, die kontinuierlich Lösungen für komplexe Probleme wie das Expression Problem bereitstellen [7].

4 Datenzentrierter Ansatz mit Generics

4.1 Beschreibung und Idee

Der datenzentrierte Ansatz adressiert das Expression Problem durch die Entwicklung eines Systems aus generischen Schnittstellen und Klassen, welches ohne Neukompilation bestehender Komponenten erweitert werden kann. Die Nutzung von Generics zur Definition abstrakter Datentypen, deren Spezifikation erst zur Laufzeit erfolgt, resultiert in einer sowohl flexiblen als auch zukunftssicheren Architektur. Die Integration neuer Datentypen und Funktionen ist ohne Beeinträchtigung existierender Strukturen möglich [8]. Die Entkopplung von Datenstrukturen und Algorithmen durch Generics steigert die Typensicherheit und ermöglicht eine flexiblere Datenhandhabung. Dies führt zu saubererem, wartbarerem und anpassungsfähigerem Code, der Änderungen am Datenmodell besser standhält [9].

4.2 Vorteile und Überwindung der Limitierungen des klassen-basierten Ansatzes

Die Vorteile des datenzentrierten Ansatzes resultieren direkt aus der Entkopplung von Daten und Algorithmen sowie der Nutzung von Generics zur Laufzeitspezifikation:

- **Modularität und Erweiterbarkeit**: Neue Datentypen und Operationen lassen sich hinzufügen, ohne bestehende Klassen oder Methoden zu modifizieren, was die Anwendung modular und leicht erweiterbar macht [8].
- **Vermeidung von Code-Duplizierung**: Generische Klassen und Methoden verringern die Notwendigkeit, ähnlichen Code für unterschiedliche Datentypen wiederholt zu schreiben, was die Wartbarkeit steigert und Fehlerquellen reduziert [7].

- **Typensicherheit**: Durch die Definition von Typen zur Compile-Zeit wird die Typensicherheit verbessert, wodurch viele Fehler frühzeitig erkannt und vermieden werden [6].
- **Wiederverwendbarkeit**: Die einfache Implementierung neuer generischer Klassen steigert die Flexibilität und Wiederverwendbarkeit des Codes, was die schnelle Anpassung an neue Anforderungen ermöglicht [7].

4.3 Implementierung in TypeScript

In diesem Abschnitt erläutern wir die Implementierung eines datenzentrierten Ansatzes in TypeScript. Dieser Ansatz zeigt, wie durch den Einsatz von Generics und dem CRTP[2] Design Pattern eine flexible und erweiterbare Struktur geschaffen werden kann, um das Ausdrucksproblem zu lösen.

Zunächst definieren wir die grundlegenden Interfaces und Klassen, um die Basisstruktur für verschiedene Post-Typen zu schaffen und erweitern diese dann, um zusätzliche Funktionalitäten hinzuzufügen.

```typescript
interface IPost<C extends IPost<C>> {
    display(): void;
}

class TextPost<C extends IPost<C>> implements IPost<C> {
    content: string;

    constructor(content: string) {
        this.content = content;
    }

    display(): void {
        console.log(`${this.content}`);
    }
}

class CombinePost<C extends IPost<C>> implements IPost<C> {
    post1, post2: IPost<any>;

    constructor(post1: IPost<any>, post2: IPost<any>) {
        this.post1 = post1;
        this.post2 = post2;
    }

    display(): void {
        console.log("Combined Post:");
        this.post1.display();
        this.post2.display();
    }
}

const textPost1 = new TextPost("Hello World!");
const textPost2 = new TextPost("This is a text post");
const combinePost = new CombinePost(textPost1, textPost2);

textPost1.display(); // Ausgabe: Hello World!
textPost2.display(); // Ausgabe: This is a text post
combinePost.display(); // Ausgabe: Combined Post: Hello World! This is a text post
```

Listing 1: Implementierung der `Post`— `TextPost` und `CombinePost` Klassen in TypeScript

Erklärung: Das Interface `IPost` definiert die `display()`-Methode für alle Posts. Die Klasse `TextPost` implementiert dieses Interface und ermöglicht die Anzeige von Text-Posts. `CombinePost` kombiniert zwei `IPost`-Instanzen und zeigt beide an.

[2]CRTP: Curiously Recurring Template Pattern

Um die Funktionalität zum Teilen von Posts zu implementieren, fügen wir das Interface `Shareable` und zugehörige Klassen hinzu. Dies erlaubt es, Posts über Social-Media-Plattformen oder andere Kommunikationskanäle zu verbreiten.

```
interface IShareable<C extends IShareable<C>> extends IPost<C> {
    share(): void;
}

class ShareableTextPost<C extends IShareable<C>>
extends TextPost<C> implements IShareable<C> {
    share(): void {
        console.log("Sharing text post: " + this.content);
    }
}

class ShareableCombinePost<C extends IShareable<C>>
extends CombinePost<C> implements IShareable<C> {
    share(): void {
        console.log("Sharing combined post:");
        this.post1.display();
        this.post2.display();
    }
}
```

Listing 2: Implementierung des `Shareable` Interfaces für teilbare Posts

Erklärung: Das Interface `IShareable` erweitert das `IPost`-Interface und fügt eine `share()`-Methode hinzu, die das Teilen von Inhalten ermöglicht. Die Klasse `ShareableTextPost` implementiert dieses Interface, indem sie von `TextPost` erbt, was das Teilen von Text-Posts ermöglicht. Ebenso erweitert die Klasse `ShareableCombinePost` die `CombinePost`-Klasse und implementiert das `IShareable`-Interface, wodurch kombinierte Posts geteilt werden können.

Um die Generizität festzulegen und instanziierbare Klassen zu schaffen, verwenden wir Fixpunktklassen. Diese Technik ist entscheidend für die Typsicherheit und Flexibilität der Architektur.

```
interface IShareablePostFix extends IShareable<IShareablePostFix> {}

class ShareableTextPostFix extends ShareableTextPost<IShareablePostFix>
implements IShareablePostFix {
    constructor(content: string) {
        super(content);
    }
}

class ShareableCombinePostFix extends ShareableCombinePost<IShareablePostFix>
implements IShareablePostFix {
    constructor(post1: IPost<any>, post2: IPost<any>) {
        super(post1, post2);
    }
}
```

Listing 3: Definition von Fixpunktklassen zur Sicherstellung der Generizität

Erklärung: Das Interface `IShareablePostFix` setzt `IShareable` als Typparameter auf sich selbst fest, um die Typensicherheit und Konsistenz innerhalb des Typsystems zu gewährleisten. Dadurch wird sichergestellt, dass die Share-Funktionalität korrekt typisiert ist und keine Typfehler zur Laufzeit auftreten. Die Klassen `ShareableTextPostFix` und `ShareableCombinePostFix` spezifizieren die konkreten Typen für Text-Posts und kombinierte Posts, wodurch sie instanzierbar werden. Dies ermöglicht eine klare und präzise Verwendung der Typen in der Anwendung, was insbesondere bei der Erweiterung und Wartung des Codes von Vorteil ist.

Zum Abschluss demonstrieren wir die Nutzung der definierten Klassen, um die Funktionalität und Interoperabilität unseres Systems zu zeigen.

```
const post1Fix = new ShareableTextPostFix("Hello, world!");
const post2Fix = new ShareableTextPostFix("This is another text post");
const combinedPostFix = new ShareableCombinePostFix(post1Fix, post2Fix);

post1Fix.display(); // Ausgabe: Hello World!
post1Fix.share(); // Ausgabe: Sharing text post: Hello World!

post2Fix.display(); // Ausgabe: This is a text post
post2Fix.share(); // Ausgabe: Sharing text post: This is a text post

combinedPostFix.display(); // Ausgabe: Combined Post: Hello World! This is a text post
combinedPostFix.share(); // Ausgabe: Sharing combined post: Combined Post: Hello World!
    This is a text post
```

Listing 4: Beispielhafte Nutzung der implementierten Klassen

Diese Implementierung zeigt, wie TypeScript genutzt werden kann, um ein flexibles und erweiterbares Typsystem zu schaffen. Der datenzentrierte Ansatz mit Generics ermöglicht eine Architektur, die sowohl Typensicherheit als auch Wiederverwendbarkeit gewährleistet. Neue Post-Typen und Methoden können hinzugefügt werden, ohne den bestehenden Code wesentlich verändern zu müssen, was den Kern des Ausdrucksproblems darstellt.

5 Operationszentrierter Ansatz mit Generics

5.1 Konzept und Erklärung

Der operationszentrierte Ansatz nutzt Generics, um eine klare Trennung zwischen Datenstrukturen und den darauf anwendbaren Operationen zu schaffen. Generische Schnittstellen definieren Operationen wie Posten, Teilen und Validieren, die auf verschiedene Datentypen anwendbar sind. Neue Operationen können hinzugefügt werden, ohne bestehende Datenstrukturen zu ändern, was die Modularität und Wiederverwendbarkeit des Codes fördert [8]. Zusätzlich erhöhen Generics die Typsicherheit und verbessern die Wartbarkeit des Systems [6].

Im Kern liegt dieser Ansatz in der Erweiterung des Visitor-Patterns. Neue Datentypen oder Operationen können hinzugefügt werden, ohne bestehende Datenstrukturen zu modifizieren. Generics ermöglichen eine flexible und erweiterbare Architektur, indem sie in die Definition der Besucher und der zu besuchenden Elemente integriert werden [9].

5.2 Anwendung von Generics im Visitor-Pattern

Im operationszentrierten Ansatz mit Generics werden generische Typen in den Visitor-Schnittstellen und den besuchten Elementen verwendet. Dies ermöglicht es Besuchern, spezifische Methoden für neue Datentypen zu implementieren, ohne die bestehenden Klassen zu verändern [3]. In Beispiel 1 könnte das Hinzufügen einer neuen Funktion für Shareable-Posts durch die Erweiterung der Visitor-Schnittstellen erfolgen, sodass diese neue Funktion ohne Änderungen an den bestehenden Text-, Bild- oder Video-Beiträgen integriert werden kann.

5.3 Vorteile und Nachteile

Der operationszentrierte Ansatz mit Generics ermöglicht das Hinzufügen neuer Operationen ohne Änderungen am bestehenden Code, was die Erweiterbarkeit und Anpassungsfähigkeit des Systems fördert [6]. Die Trennung von Operationen und Daten verbessert die Modularität und Wartbarkeit des Codes [3]. Generics erhöhen die Typensicherheit durch Typüberprüfung zur Compile-Zeit und reduzieren potenzielle Laufzeitfehler [7].

Dieser Ansatz bringt jedoch eine erhöhte Komplexität bei der Implementierung und dem Verständnis des Codes mit sich, besonders für Entwickler mit weniger Erfahrung in Generics. Die korrekte Anwendung von Generics im Visitor-Pattern erfordert mehr Fachwissen und kann die Einarbeitungszeit verlängern [4].

6 Schluss

6.1 Zusammenfassung

In dieser Ausarbeitung wurde das Expression Problem von seinen historischen Wurzeln bis zu modernen Lösungen mit Generics in TypeScript untersucht. Zu Beginn wurden die Arbeiten von John Reynolds [5] und die Definition von Philip Wadler [9] erläutert, um die Herausforderung der Erweiterbarkeit von Datentypen und Operationen ohne Neukompilierung des Codes zu verdeutlichen.

Klassische Ansätze wie der klassenbasierte Ansatz und das Visitor-Pattern wurden vorgestellt [2], gefolgt von einer Analyse der Vorteile von Generics für Typensicherheit und Wiederverwendbarkeit [6]. Es wurde gezeigt, wie TypeScript aufgrund seiner starken Unterstützung für Generics ausgewählt wurde [7]. Der datenzentrierte Ansatz ermöglicht eine flexible und erweiterbare Architektur, während der operationszentrierte Ansatz die Modularität verbessert [8]. Insgesamt bietet TypeScript mit Generics effektive Lösungen für das Expression Problem, indem Flexibilität und Typensicherheit vereint werden.

6.2 Fazit

Die Analyse des *Expression Problems* und die Implementierung der Lösungen mit Generics in TypeScript haben gezeigt, dass diese Ansätze entscheidende Vorteile bieten. Generics ermöglichen die Entwicklung flexibler und typsicherer Software, die sowohl modular als auch wartbar ist. TypeScript hat sich als geeignete Sprache erwiesen, um diese Konzepte effektiv umzusetzen.

Die vorgestellten Ansätze zur Lösung des *Expression Problems* zeigen, dass durch den Einsatz von Generics die Erweiterbarkeit und Wiederverwendbarkeit von Code erheblich verbessert werden können. Diese Techniken bieten eine robuste Grundlage für die Entwicklung zukünftiger Softwareprojekte. Nachteile wie erhöhte Komplexität bei der Implementierung können durch sorgfältige Planung und Design minimiert werden.

Insgesamt bieten die Lösungen für das *Expression Problem* wertvolle Ansätze für die Softwareentwicklung und tragen zur Schaffung von zukunftssicheren Anwendungen bei.

Literatur

[1] William Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages (FOOL), REX School/Workshop*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178, Noordwijkerhout, The Netherlands, 1990. Springer Berlin Heidelberg.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994.

[3] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[4] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote reuse. In *European Conference on Object-Oriented Programming (ECOOP)*, Department of Computer Science, Rice University, 1998. A preliminary version of this paper appeared in the European Conference on Object-Oriented Programming.

[5] John C. Reynolds. *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*, pages 309–317. Springer New York, New York, NY, 1978. Originally presented at the IFIP Working Group 2.1 conference on New Directions in Algorithmic Languages, Munich, 1975.

[6] Nathan Rozentals. *Mastering TypeScript*. Packt Publishing, 2019.

[7] Basarat Ali Syed. *TypeScript Deep Dive*. 2019. Zugriff am 01. Juli 2024.

[8] Mads Torgersen. The expression problem revisited: Four new solutions using generics. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pages 123–143, Berlin, Heidelberg, 2004. Springer. SourceDBLP.

[9] Philip Wadler. The expression problem. Online, November 1998. Der Artikel ist auf folgender URL erhältlich: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (zuletzt besucht am 18. Juli 2024).