DHBW
Duale Hochschule
Baden-Württemberg

Advanced Software Engineering

Angewandtes Projektmanagement

Studiengang Informatik

Duale Hochschule Baden-Württemberg Karlsruhe

Matej Evic

Eingereicht am: 11.06.2025
Matrikelnummer, Kurs: 7962582, TINF22B2
Betreuer an der DHBW: Michael Vetter

Inhalt

1

Einführung

1.1 Übersicht über die Applikation

Die Applikation ist ein Cache-Simulator, mit dem man nachvollziehen kann, wie ein CPU-Cache arbeitet. Das Tool zeigt, wie Daten zwischen Hauptspeicher (RAM) und Cache verschoben werden und wie verschiedene Caching-Strategien wie LRU, LFU oder FIFO das Verhalten und die Performance beeinflussen.

Der Simulator hilft dabei, Cache-Algorithmen und typische Probleme wie Cache Misses oder Verdrängungen besser zu verstehen. Die Anwendung richtet sich an Entwickler, Studierende und alle, die sich für Rechnerarchitektur interessieren.

Man kann verschiedene Cache-Arten simulieren, darunter Direct-Mapped, Fully-Associative und N-Way-Set-Associative. Die Software liefert detaillierte Statistiken wie Trefferquote, Zugriffe und Verdrängungen. Über die Kommandozeile kann man entweder Adressen manuell eingeben oder eine automatische Simulation starten.

1.2 Wie startet man die Applikation?

Voraussetzungen:
- Java 17 oder neuer
- Maven (Build-Tool)

Schritt-für-Schritt-Anleitung:

1. Projekt herunterladen:

```
git clone https://github.com/MatejEvc/CacheSoftEng.git
cd CacheSoftEng
```

2. Applikation bauen:

```
mvn clean install
```

Dadurch entsteht eine ausführbare JAR-Datei im `target`-Ordner.

3. Applikation starten:

```
java -jar target/Cache-1.0-SNAPSHOT.jar
```

Alternativ kannst man die Main-Klasse `com.cacheproject.CacheApp` direkt in der IDE (z.B. IntelliJ oder VS Code) starten.

Nach dem Start siehst du ein Menü mit folgenden Optionen:

1. Access memory address – Speicheradresse eingeben und Zugriff simulieren
2. Display cache state – Den aktuellen Zustand des Caches als ASCII-Tabelle anzeigen
3. Run simulation – Eine kleine Demo mit vorgegebenen Adressen durchlaufen lassen
4. Show statistics – Aktuelle Statistiken anzeigen (Hits, Misses, Verdrängungen usw.)
5. Reset statistics – Statistiken zurücksetzen
6. Exit – Programm beenden

1.3 Wie testet man die Applikation?

Voraussetzungen:
- Java 17+
- Maven
- (Optional) Eine IDE wie IntelliJ, falls man Tests manuell ausführen möchte

Testing:

1. Unit-Tests automatisch ausführen:

   ```
   mvn test
   ```

   Alle Kernfunktionen und Logik werden damit automatisch geprüft.

2. Manueller Test über die Kommandozeile: Starte die Anwendung wie oben beschrieben. Wähle im Menü zum Beispiel Option 1 und gib verschiedene Adressen ein, um das Verhalten des Caches zu sehen. Mit Option 3 kannst du eine kleine Simulation durchlaufen lassen. Mit Option 4 bekommst du die aktuellen Statistiken angezeigt.

2

Clean Architecture

2.1 Was ist Clean Architecture?

Clean Architecture ist ein Architekturstil, der Software in Schichten unterteilt, wobei die Domain-Logik im Zentrum steht. Die Grundprinzipien sind:

- Unabhängigkeit der Kernlogik von Frameworks, Datenbanken und UI
- Äußere Schichten dürfen innere Schichten kennen, aber nie umgekehrt
- Testbarkeit der Business-Logik ohne Infrastruktur

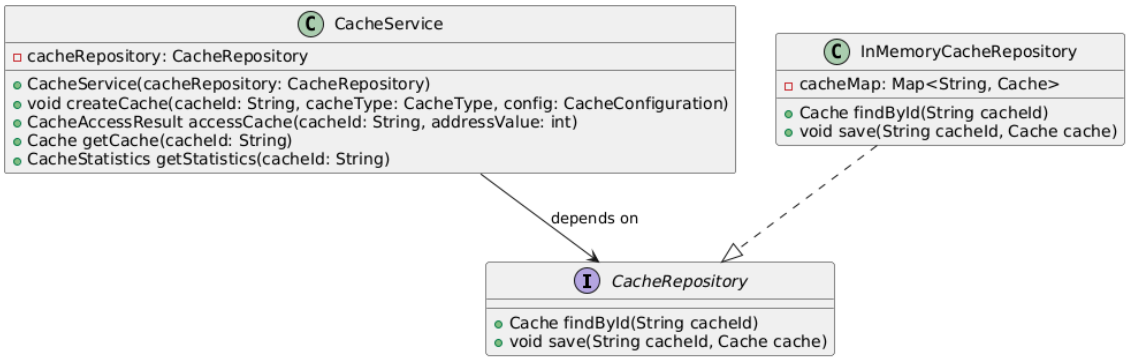
2.2 Analyse der Dependency Rule

Im Sinne der Clean Architecture gilt die Dependency Rule: Abhängigkeiten zeigen immer von außen nach innen, nie umgekehrt. Das bedeutet, dass z.B. die Service-Schicht nur von Interfaces der Domain- oder Application-Schicht abhängt und nicht direkt von Implementierungen der Infrastruktur.

2.2.1 Positiv-Beispiel

Ein gutes Beispiel im Projekt ist die Klasse CacheService. Sie steht im Application Layer und hängt ausschließlich vom Interface CacheRepository ab, nicht von einer konkreten Implementierung wie InMemoryCacheRepository. Die konkrete Repository-Implementierung wird erst zur Laufzeit (z.B. im Main-Programm) injiziert.

UML-DIAGRAMM



ANALYSE DER ABHÄNGIGKEITEN

- Von wem hängt die Klasse ab?
  - CacheService hängt ausschließlich vom Interface CacheRepository ab.
  - Es gibt keine direkte Abhängigkeit zu einer Infrastrukturklasse oder zu Framework-Code.

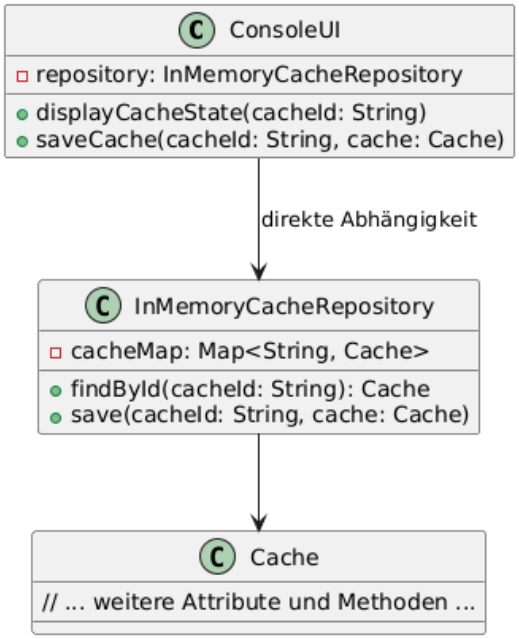
- Wer hängt von der Klasse ab?
  - Die Präsentationsschicht (ConsoleUI) nutzt den CacheService, um Cache-Funktionen auszuführen.
  - Die Main-Klasse (CacheApp) erstellt und verbindet die Komponenten.

- Dependency Rule:
  - Die Abhängigkeit zeigt klar von außen (UI, Main) nach innen (Service, Repository-Interface).
  - Die Implementierung des Repositories (InMemoryCacheRepository) kann beliebig ausgetauscht werden, ohne dass der Service-Code angepasst werden muss.
  - Die Kernlogik bleibt unabhängig von technischen Details der Infrastruktur.

2.2.2 Negativ-Beispiel

Ein Negativbeispiel für die Dependency Rule wäre eine Präsentationsklasse wie ConsoleUI, die direkt von einer konkreten Infrastrukturklasse wie InMemoryCacheRepository abhängt. Das verletzt die Clean Architecture, weil eine äußere Schicht (UI) direkt von einer anderen äußeren Schicht (Infrastruktur) abhängig ist, anstatt nur mit Interfaces oder Services aus der Application-Schicht zu kommunizieren.

UML-Diagramm



Analyse der Abhängigkeiten:

- Von wem hängt ConsoleUI ab?
  - Direkt von der konkreten Klasse InMemoryCacheRepository aus der Infrastruktur-Schicht.
- Wer hängt von ConsoleUI ab?
  - Die Main-Klasse (CacheApp) oder andere Teile der Präsentationsschicht.
- Bezug zur Dependency Rule:
  - Die Abhängigkeit verläuft von außen (Präsentation) zu einer anderen äußeren Schicht (Infrastruktur) – das verstößt gegen die Regel, dass Abhängigkeiten immer nur nach innen zeigen dürfen.
  - Änderungen an der Infrastruktur (z.B. Wechsel von InMemory zu DatabaseRepository) würden Anpassungen in der UI erzwingen.
  - Die Testbarkeit leidet, weil die UI nicht mehr einfach mit Mocks oder anderen Implementierungen getestet werden kann.

2.3 Analyse der Schichten

```
App
├── domain/          # Kernlogik (Cache, Replacement Policies)
├── infrastructure/  # Implementierungen (InMemoryRepository)
├── application/     # Services (CacheService)
└── presentation/    # UI (ConsoleUI)
```

2.3.1 Schicht: Domain Layer (Kernlogik)

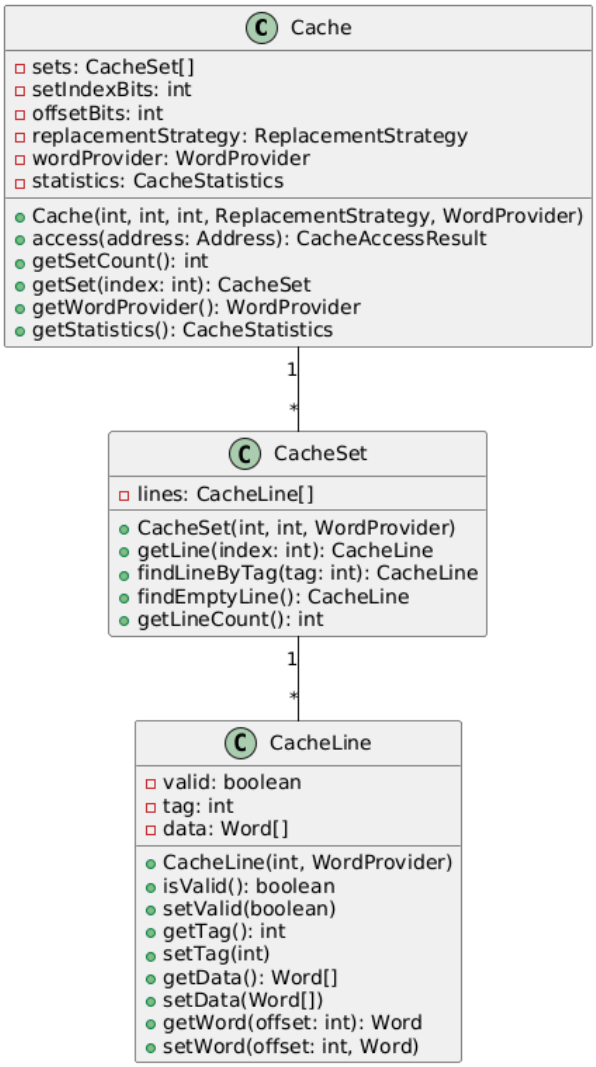
Die Domain-Schicht bildet das Herzstück der Clean Architecture. Hier liegen die zentralen Geschäftsregeln und die eigentliche Logik, unabhängig von technischen Details wie Datenhaltung oder Benutzeroberfläche.

Beispiel-Klasse: Cache

Aufgabe: Die Klasse Cache steuert das komplette Verhalten des Caches. Sie verwaltet die Cache-Sets, führt Zugriffe durch, entscheidet über Hits und Misses und steuert die Ersetzungsstrategie. Außerdem hält sie die Statistiken und kennt die Konfiguration.

Einordnung: Cache ist eine reine Domänenklasse. Sie kennt keine Details über UI, Frameworks oder Datenbanken und ist nur von anderen Domain-Klassen abhängig.

UML:



2.3.2 Schicht: Application Layer (Service/Adapter)

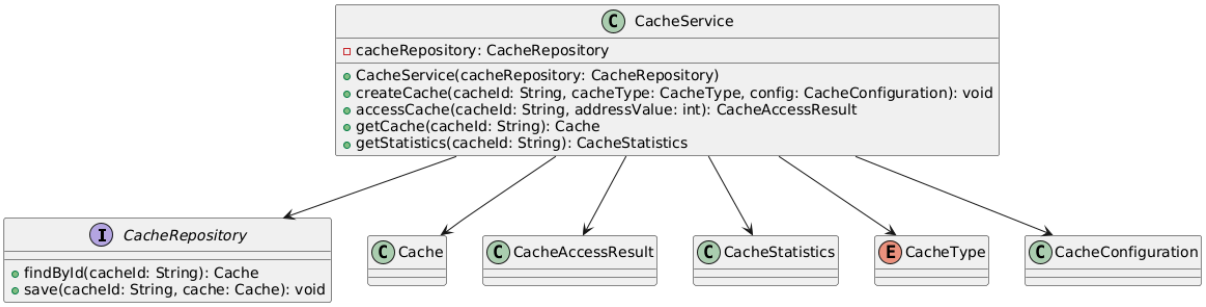
Die Application-Schicht vermittelt zwischen der Präsentation und der Domain. Hier werden Anwendungsfälle koordiniert, aber keine Geschäftsregeln definiert.

Beispiel-Klasse: CacheService

Aufgabe: CacheService nimmt Anfragen entgegen (z.B. von der UI), ruft die passenden Methoden der Domain-Klassen auf und gibt Ergebnisse zurück. Er kapselt die Abläufe rund um das Anlegen, Zugreifen und Auslesen von Caches. Die Klasse arbeitet nur mit Interfaces wie CacheRepository und bleibt so unabhängig von konkreten Implementierungen.

Einordnung: CacheService ist klassisch im Application Layer angesiedelt. Er kennt die Domain-Schicht, aber keine Details der Präsentation oder Infrastruktur.

UML:



2.3.3 Schicht: Infrastructure Layer

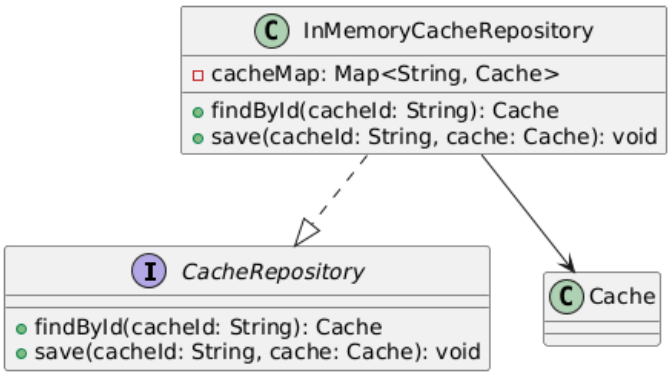
Die Infrastructure-Schicht ist für die Anbindung an externe Systeme und technische Details zuständig. Sie stellt konkrete Implementierungen für Interfaces bereit, die in den inneren Schichten definiert wurden.

Beispiel-Klasse: InMemoryCacheRepository

Aufgabe: InMemoryCacheRepository speichert und verwaltet Cache-Instanzen im Speicher (HashMap). Sie implementiert das Interface CacheRepository, das in der Application-Schicht genutzt wird. Die Domain- und Application-Schichten wissen nicht, wie und wo die Daten gespeichert werden – sie kennen nur das Interface.

Einordnung: InMemoryCacheRepository gehört zur Infrastructure-Schicht, weil sie technische Details kapselt und die eigentliche Speicherung übernimmt. Sie hängt von keiner inneren Schicht ab, sondern implementiert nur deren Vorgaben.

UML:



2.3.4 Schicht: Presentation Layer

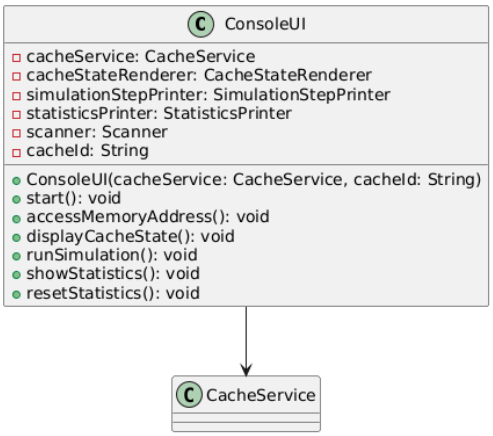
Die Presentation-Schicht ist für die Interaktion mit dem Nutzer zuständig. Sie nimmt Eingaben entgegen, zeigt Ausgaben an und ruft die Application-Schicht auf, um Aktionen auszuführen.

Beispiel-Klasse: ConsoleUI

Aufgabe: ConsoleUI ist die Kommandozeilen-Oberfläche der Anwendung. Sie zeigt ein Menü an, verarbeitet Nutzereingaben, ruft Methoden des CacheService auf und gibt die Ergebnisse aus. Die Klasse weiß nichts über die interne Funktionsweise des Caches oder der Speicherung – sie arbeitet nur mit dem Service.

Einordnung: ConsoleUI ist Teil der Presentation-Schicht, weil sie die Schnittstelle zwischen Nutzer und Anwendung darstellt. Sie ist von der Application-Schicht abhängig, aber nicht von Domain oder Infrastructure.

UML:



3

Kapitel 3: SOLID

3.1 Analyse Single-Responsibility-Principle (SRP)

Das Single-Responsibility-Principle (SRP) sagt, dass eine Klasse immer nur eine Aufgabe bzw. einen Grund zur Änderung haben sollte. Das macht den Code übersichtlicher, leichter testbar und besser wartbar.

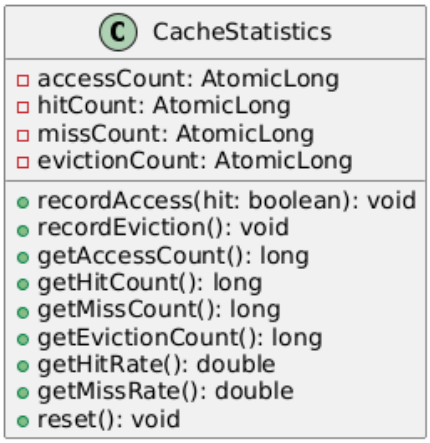
Im aktuellen Stand des Projekts gibt es keine Klasse, die das SRP verletzt. Daher werden hier zwei Klassen vorgestellt, die das SRP vorbildlich einhalten.

3.1.1 Positiv-Beispiel

Klasse: CacheStatistics

Beschreibung: CacheStatistics kümmert sich nur um das Zählen und Berechnen von Zugriffen, Treffern, Misses und Verdrängungen im Cache. Sie speichert die reinen Statistikdaten und bietet Methoden, um diese zu aktualisieren oder auszulesen. Es gibt keine Logik für Cache-Zugriffe, keine Ausgabe oder andere fachfremde Aufgaben – genau so, wie es das SRP vorsieht.

UML:



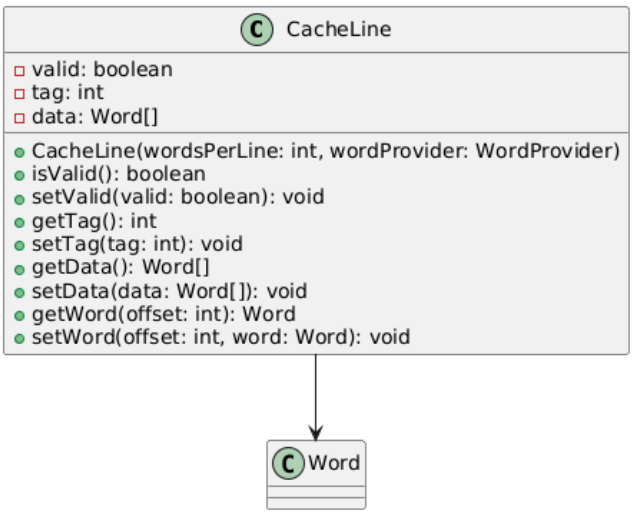
Die Klasse hat genau eine Verantwortung: Statistik führen. Sie ändert sich nur, wenn sich das Zählen oder die Berechnung der Statistik ändert. Alles andere (Cache-Logik, Darstellung, Speicherung) ist ausgelagert.

3.1.2 2. Positiv-Beispiel

Klasse: CacheLine

Beschreibung: Die Klasse CacheLine repräsentiert eine einzelne Cache-Zeile. Sie ist nur für die Verwaltung ihres eigenen Zustands zuständig: Tag, Valid-Flag und die gespeicherten Daten (Words). Sie enthält keine Logik für den Cache-Zugriff, keine Statistik und keine Darstellung.

UML:



Auch CacheLine hat nur eine Aufgabe: Sie verwaltet den Zustand einer einzelnen Line im Cache. Änderungen an der Art, wie Daten gespeichert werden, betreffen nur diese Klasse.

3.2 Analyse Open-Closed-Principle (OCP)

Das Open-Closed-Principle (OCP) besagt: Eine Klasse soll offen für Erweiterung, aber geschlossen für Modifikation sein. Das heißt, das Verhalten einer Klasse kann durch Vererbung, Interfaces oder Komposition erweitert werden, ohne dass der bestehende Code verändert werden muss. Dadurch bleibt das System stabil und flexibel für neue Anforderungen.

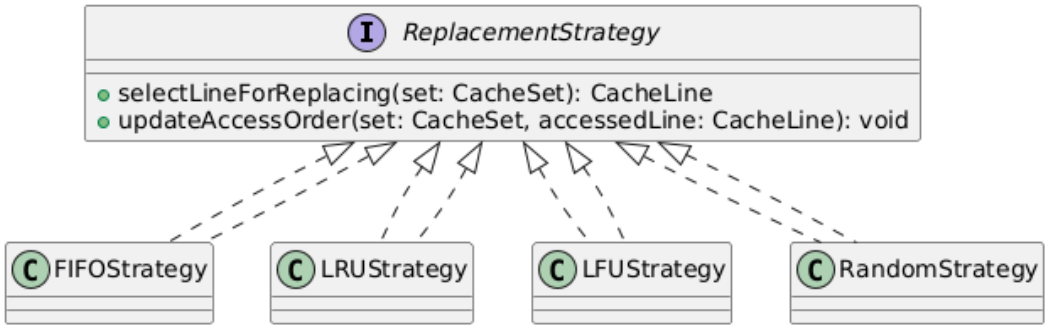
Im aktuellen Stand des Projekts gibt es keine Klasse, die das OCP verletzt. Daher werden hier zwei Klassen vorgestellt, die das OCP vorbildlich umsetzen.

3.2.1 Positiv-Beispiel

Klasse: ReplacementStrategy und ihre Implementierungen (FIFOStrategy, LRUStrategy, LFUStrategy, RandomStrategy)

Beschreibung: Im Projekt gibt es das Interface ReplacementStrategy, das die Methoden selectLineForReplacing und updateAccessOrder definiert. Für jede Ersetzungsstrategie (FIFO, LRU, LFU, Random) gibt es eine eigene Klasse, die dieses Interface implementiert. Der Cache verwendet nur das Interface. Neue Strategien können einfach hinzugefügt werden, indem man eine neue Klasse schreibt, die das Interface implementiert – der bestehende Code im Cache bleibt unverändert.

UML:



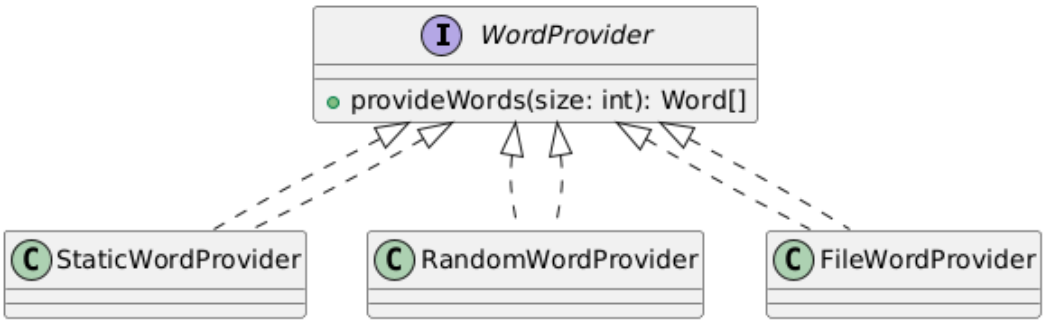
Analyse:
- Der Cache ist offen für Erweiterung: Neue Strategien können hinzugefügt werden, ohne dass der Cache-Code geändert werden muss.
- Der Cache ist geschlossen für Modifikation: Die bestehende Logik bleibt stabil, auch wenn neue Strategien hinzukommen.
- Das ist sinnvoll, weil man so beliebig viele Ersetzungsstrategien anbieten kann, ohne dass man den Cache oder andere Teile des Systems anfassen muss.

3.2.2 2. Positiv-Beispiel

WordProvider und ihre Implementierungen

Beschreibung: Das Interface WordProvider definiert die Methode provideWords(int size). Es gibt verschiedene Implementierungen, z.B. StaticWordProvider, RandomWordProvider und FileWordProvider. Der Cache und die zugehörigen Klassen verwenden immer nur das Interface. Neue Arten, wie Words bereitgestellt werden, können einfach als neue Klasse implementiert werden – der bestehende Code bleibt unverändert.

UML:



Analyse:
- Das System ist offen für Erweiterung: Es können jederzeit neue Arten von Word-Quellen ergänzt werden.
- Der Code ist geschlossen für Modifikation: Weder der Cache noch andere Nutzer des WordProvider-Interfaces müssen angepasst werden.
- Das ist praktisch, weil so z.B. auch ein Datenbank-Provider für Words ergänzt werden kann, ohne dass bestehende Funktionalität geändert werden muss.

3.3 Analyse Liskov Substitution Principle (LSP)

Das Liskov Substitution Principle (LSP) sagt: Eine abgeleitete Klasse muss überall dort einsetzbar sein, wo ihre Basisklasse erwartet wird. Das bedeutet, dass Subklassen sich wie die Oberklasse verhalten und keine unerwarteten Fehler oder Seiteneffekte verursachen dürfen.

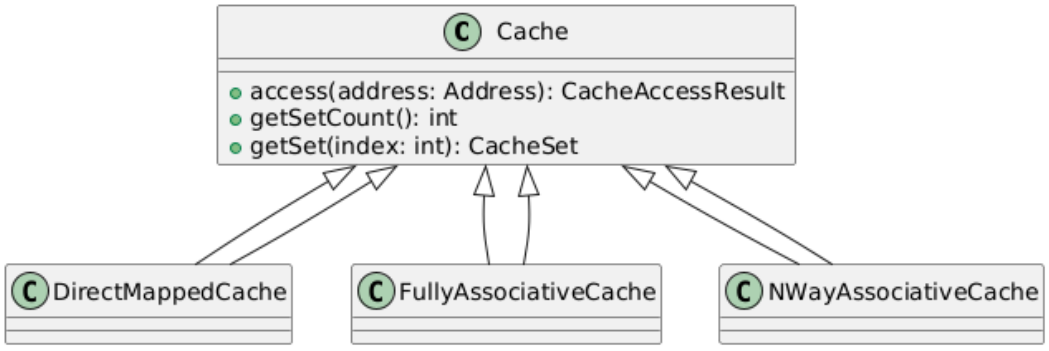
Im aktuellen Stand des Projekts gibt es keine Klasse, die das LPS verletzt. Daher werden hier zwei Klassen vorgestellt, die das LSP vorbildlich umsetzen.

3.3.1 Positiv-Beispiel

Klassen:
- Cache (Basisklasse)
- DirectMappedCache, FullyAssociativeCache, NWayAssociativeCache (Subklassen)

Beschreibung: Alle drei Cache-Typen erben von der Basisklasse Cache. Sie überschreiben keine Methoden mit anderem Verhalten, sondern spezialisieren nur die Konstruktion (z.B. Anzahl Sets oder Assoziativität). Überall, wo ein Cache erwartet wird (z.B. im Service, Repository oder Factory), kann auch ein DirectMappedCache, FullyAssociativeCache oder NWayAssociativeCache verwendet werden – ohne dass das System anders funktioniert.

UML:



Begründung:
- Jede Subklasse kann überall da eingesetzt werden, wo ein Cache gebraucht wird.
- Es gibt keine Methoden, die in den Subklassen plötzlich Fehler werfen oder sich anders verhalten als in der Basisklasse.
- Das Prinzip ist erfüllt, weil die Vererbung korrekt genutzt wird.

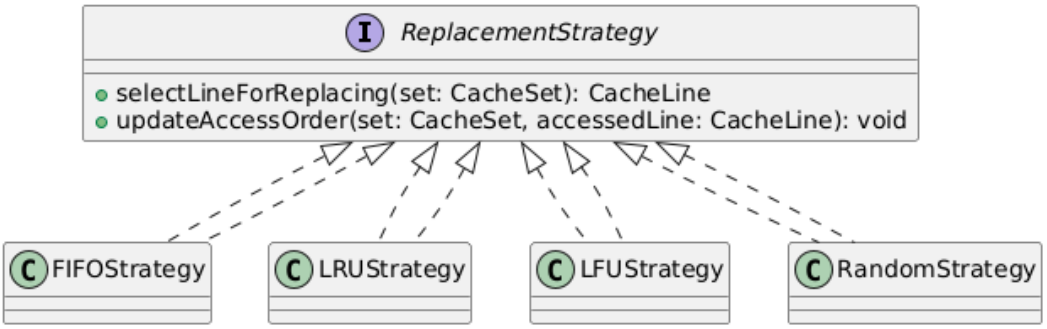
3.3.2 2. Positiv-Beispiel

Klassen:
- Interface ReplacementStrategy
- Implementierungen: FIFOStrategy, LRUStrategy, LFUStrategy, RandomStrategy

Beschreibung: Das Interface ReplacementStrategy definiert zwei Methoden:
- `CacheLine selectLineForReplacing(CacheSet set)`
- `void updateAccessOrder(CacheSet set, CacheLine line)`

Jede Ersetzungsstrategie implementiert diese Methoden auf ihre spezifische Weise, aber alle halten sich an den Vertrag des Interfaces. Der Cache kann jede Strategie verwenden, ohne dass sich sein Verhalten unerwartet ändert.

UML:



Begründung:
- Austauschbarkeit: Der Cache kann beliebige ReplacementStrategy-Implementierungen verwenden, ohne seinen Code ändern zu müssen.
- Konsistentes Verhalten: Alle Strategien garantieren, dass selectLineForReplacing eine gültige CacheLine zurückgibt und updateAccessOrder den Zugriff protokolliert.
- Keine Implementierung wirft unerwartete Exceptions oder verletzt die Semantik der Methoden.

4

Weitere Prinzipien

4.1 Analyse GRASP: Geringe Kopplung

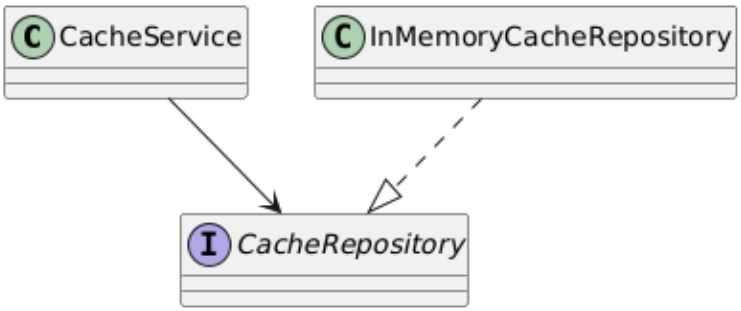
Geringe Kopplung bedeutet, dass Klassen möglichst unabhängig voneinander sind und nur über klar definierte Schnittstellen kommunizieren.

4.1.1 Positiv-Beispiel

CacheService mit CacheRepository-Interface

Klassen:
- CacheService (Application Layer)
- CacheRepository (Interface)
- InMemoryCacheRepository (Infrastructure Layer)

UML:



Aufgabenbeschreibung:
- CacheService verwaltet die Geschäftslogik für Cache-Operationen.
- CacheRepository definiert die Schnittstelle für die Datenhaltung.
- InMemoryCacheRepository implementiert die Speicherung im Speicher.

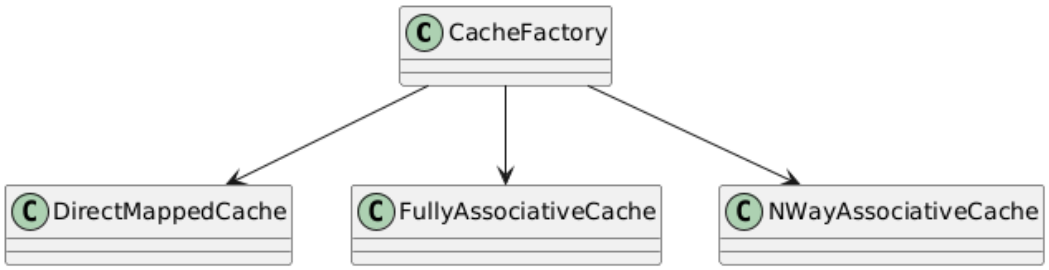
Begründung für geringe Kopplung:
- CacheService hängt nur vom Interface CacheRepository ab, nicht von konkreten Implementierungen.
- Die Datenhaltung kann einfach ausgetauscht werden (z.B. durch DatabaseCacheRepository), ohne den CacheService zu ändern.
- Die Kopplung ist minimal, da nur abstrakte Methoden genutzt werden.

4.1.2 Negativ-Beispiel

CacheFactory mit konkreten Cache-Klassen (https://refactoring.guru/design-patterns/abstract-factory)

Klassen:
- CacheFactory
- DirectMappedCache, FullyAssociativeCache, NWayAssociativeCache

UML:

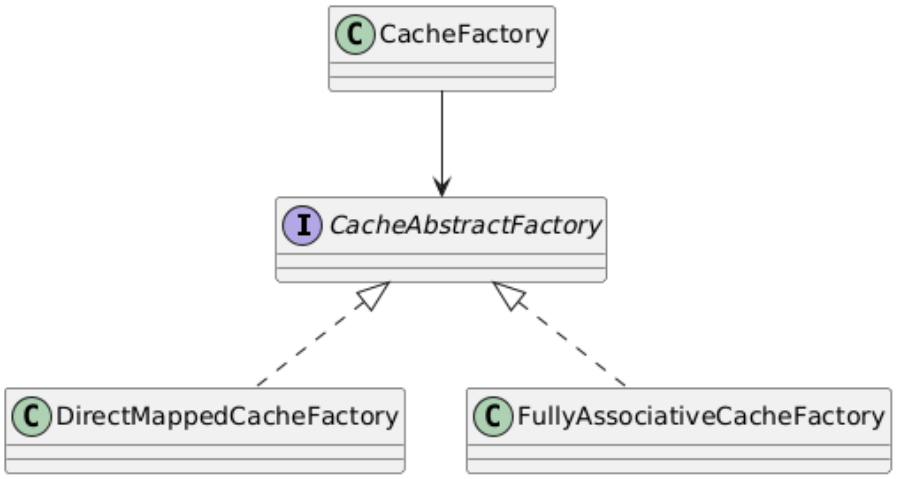


Aufgabenbeschreibung:
- CacheFactory erstellt konkrete Cache-Instanzen basierend auf dem CacheType.

Problematik der engen Kopplung:
- Die Factory kennt alle konkreten Cache-Implementierungen direkt.
- Bei Hinzufügung einer neuen Cache-Art (z.B. SetAssociativeCache) muss die Factory angepasst werden.
- Die Kopplung ist hoch, da Änderungen in den Cache-Klassen die Factory beeinflussen.

Lösungsweg zur Auflösung der Kopplung:

1. Einführung eines Abstract Factory-Patterns:



2. Jede Cache-Art hat eine eigene Factory:
   - CacheAbstractFactory definiert die Methode createCache().
   - Konkrete Factories (DirectMappedCacheFactory, etc.) implementieren die Erstellung.

3. CacheFactory delegiert an die passende Abstract Factory:

```
1     public class CacheFactory {
2     private final Map<CacheType, CacheAbstractFactory> factories;
3
4     public CacheFactory() {
5         factories = Map.of(
6             CacheType.DIRECT_MAPPED, new DirectMappedCacheFactory(),
              CacheType.FULLY_ASSOCIATIVE, new
7   FullyAssociativeCacheFactory()
8         );
9     }
10
      public Cache createCache(CacheType type, CacheConfiguration config)
11  {
12        return factories.get(type).createCache(config);
13    }
14  }
```

Listing 5 — CacheFactory

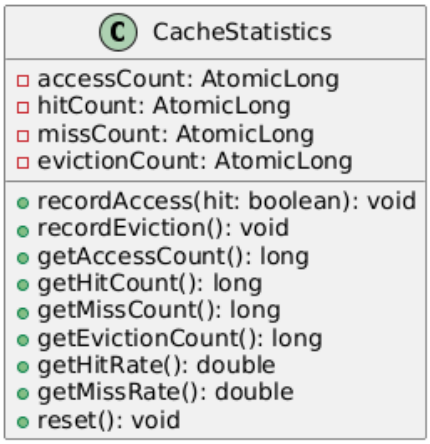
Vorteil der Lösung:
- Neue Cache-Typen können hinzugefügt werden, ohne die zentrale CacheFactory zu ändern.
- Die Kopplung wird auf die Abstract Factory reduziert, nicht auf konkrete Klassen.

4.2 Analyse GRASP: Hohe Kohäsion

4.2.1 Positiv-Beispiel

Klasse: CacheStatistics

UML-Diagramm:



Begründung für hohe Kohäsion:

- Eng fokussierte Verantwortung: Die Klasse ist ausschließlich für das Zählen und Berechnen von Cache-Statistiken zuständig. Alle Methoden und Attribute beziehen sich direkt auf dieses Kernthema:
  - Zählen von Zugriffen (accessCount)
  - Unterscheidung zwischen Hits/Misses (hitCount, missCount)
  - Verdrängungszähler (evictionCount)
  - Berechnung von Trefferquoten (getHitRate())
  - Zurücksetzen der Statistiken (reset())
- Keine fachfremden Aufgaben: Die Klasse enthält keine Logik für:
  - Cache-Zugriffe
  - Datenhaltung

  - Benutzerinteraktion
  - Formatierung der Ausgabe

- Eindeutiger Änderungsgrund: Die Klasse muss nur angepasst werden, wenn sich die Art der Statistikführung ändert (z.B. neue Metriken hinzukommen). Änderungen in anderen Teilen des Systems (z.B. Cache-Logik) beeinflussen sie nicht.

- Isolierte Funktionalität: Die Methoden arbeiten ausschließlich mit den internen Zählern und benötigen keine komplexen Abhängigkeiten zu anderen Klassen. Selbst die reset()-Methode manipuliert nur die eigenen Attribute.

- Wiederverwendbarkeit: Die Klasse könnte ohne Änderungen in anderen Cache-Implementierungen oder sogar in anderen Projekten mit Statistikbedarf eingesetzt werden.

4.3 Analyse: Don’t repeat yourself

Beispiel: Zentralisierung der Adresslogik in der Address-Klasse

Hinweis: Es gibt keinen spezifischen Commit für diese Änderung, da die Duplizierung bereits vor der ersten Versionierung des Codes behoben wurde.

Kontext: In früheren Versionen des Codes war die Logik zur Berechnung von Tag, Set-Index und Offset an mehreren Stellen dupliziert – sowohl in der Cache-Klasse als auch in der ConsoleUI. Dies verletzte das DRY-Prinzip und machte Änderungen fehleranfällig.

4.3.1 Code vor der Refaktorisierung

In Cache.java:

```
public CacheAccessResult access(int addressValue) {
    // Duplizierte Berechnung
    int offsetBits = Integer.numberOfTrailingZeros(wordsPerLine);
    int setIndexBits = Integer.numberOfTrailingZeros(setCount);
    int tag = addressValue >> (offsetBits + setIndexBits);
    int setIndex = (addressValue >> offsetBits) & ((1 << setIndexBits) -
1);
    int offset = addressValue & ((1 << offsetBits) - 1);
    // ...
}
```

Listing 6 — Cache

In ConsoleUI.java:

```
   public void printAddressDetails(int addressValue) {
    // Dieselbe Berechnung wiederholt
    int offsetBits = 2;
    int setIndexBits = 3;
    int tag = addressValue >> (offsetBits + setIndexBits);
    int setIndex = (addressValue >> offsetBits) & ((1 << setIndexBits) -
1);
    int offset = addressValue & ((1 << offsetBits) - 1);
    // ...
}
```

Listing 7 — ConsoleUI

4.3.2 Code nach der Refaktorisierung

Neue Klasse Address.java:

```
    public class Address {
    private final int value;

    public Address(int value) {
        this.value = value;
    }

    public int getTag(int setIndexBits, int offsetBits) {
        return value >>> (setIndexBits + offsetBits);
    }

    public int getSetIndex(int setIndexBits, int offsetBits) {
        return (value >>> offsetBits) & ((1 << setIndexBits) - 1);
    }

    public int getOffset(int offsetBits) {
        return value & ((1 << offsetBits) - 1);
    }
}
```

Listing 8 — CacheFactory

Neuer Code;

```
// In Cache.java
public CacheAccessResult access(Address address) {
    int setIndex = address.getSetIndex(setIndexBits, offsetBits);
    // ...
}

// In ConsoleUI.java
public void printAddressDetails(Address address) {
    int tag = address.getTag(3, 2);
    // ...
}
```

Listing 9 — Aktualisierter Code

4.3.3 Begründung und Auswirkung

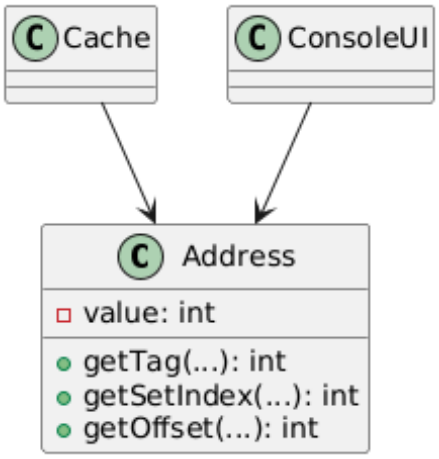
Warum wurde es geändert?

- Redundanz: Dieselbe Berechnung war an zwei Stellen dupliziert.

- Fehlerrisiko: Änderungen an der Adresslogik hätten an mehreren Stellen durchgeführt werden müssen.
- Wartbarkeit: Die Logik war schwer zu testen und zu dokumentieren.

Auswirkung der Änderung:
- Single Source of Truth: Die Address-Klasse ist jetzt die alleinige Quelle für Adressberechnungen.
- Konsistenz: Alle Stellen, die Adressen verarbeiten, nutzen dieselbe Logik.
- Flexibilität: Neue Adressformate können einfach durch Änderungen in einer Klasse eingeführt werden.

UML der Lösung:



5

Unit Tests

5.1 10 Unit Tests

siehe nächste Seite ->

Nr.	Unit Test	Beschreibung
1	„CacheConfigurationTest# differentInstances_withSameValues_areNotEqual“	„Testet, dass zwei Konfigurationen mit denselben Werten nicht als gleich gelten (Referenzungleichheit).“
2	„AddressTest#getTag_fullyAssociative“	„Überprüft korrekte Tag-Berechnung für vollassoziative Caches.“
3	„CacheLineTest#setWord_setsWordAtGivenOffset“	„Testet, ob ein Word an einer bestimmten Position in der CacheLine korrekt gesetzt wird.“
4	„CacheSetTest#findEmptyLine_returnsNullIfAllValid“	„Stellt sicher, dass keine freie Line gefunden wird, wenn alle Lines gültig sind.“
5	„CacheStatisticsTest# recordEviction_incrementsEvictionCount“	„Prüft, ob die Anzahl der Verdrängungen korrekt erhöht wird.“
6	„FIFOStrategyTest# selectLineForReplacing_returnsFirstInsertedLine“	„Testet, ob FIFO immer die älteste Line zum Ersetzen auswählt.“
7	„FileWordProviderTest# provideWords_throwsRuntimeException_onInvalidFile“	„Überprüft, ob bei ungültigem Dateipfad eine Exception geworfen wird.“
8	„CacheServiceTest# accessCache_throwsException_whenNoCache“	„Testet, dass beim Zugriff auf einen nicht existierenden Cache eine Exception geworfen wird.“
9	„DirectMappedCacheTest# access_replacesLine_whenTagDiffers“	„Prüft, ob bei Tag-Kollision im Direct-Mapped Cache eine Line ersetzt wird.“
10	„CacheStatisticsTest# recordEviction_incrementsEvictionCount“	„Prüft, ob die Anzahl der Verdrängungen korrekt erhöht wird.“

5.2 ATRIP-Analyse

5.2.1 Automatic

Die Unit-Tests im Projekt werden vollautomatisch ausgeführt. Die Integration erfolgt über Maven, sodass alle Tests mit dem Befehl

```
mvn test
```

automatisch gestartet werden. Es ist keine manuelle Interaktion notwendig: Die Tests laufen durch, prüfen das Verhalten der Klassen und liefern ein klares Ergebnis (grün/rot). Durch die Verwendung von JUnit und Mocking (z.B. Mockito) sind die Tests unabhängig von der Umgebung und können beliebig oft wiederholt werden.

5.2.2 Thorough

Thorough bedeutet, dass die Tests nicht nur Standardfälle, sondern auch Fehlerfälle, Grenzwerte und verschiedene Äquivalenzklassen abdecken

Positiv-Beispiel Test:

```
@Test
void provideWords_throwsRuntimeException_onInvalidFile() {
    FileWordProvider provider = new
FileWordProvider("notexisting_file.txt");
    assertThrows(RuntimeException.class, () -> provider.provideWords(2));
}
```

Analyse: Dieser Test prüft explizit einen Fehlerfall: Was passiert, wenn eine Datei nicht existiert? Er stellt sicher, dass der Code robust auf ungültige Eingaben reagiert und eine Exception wirft.

Begründung: Das ist professionell, weil nicht nur der Erfolgsfall, sondern auch der Fehlerfall getestet wird.

Negativ-Beispiel

Test:

```
@Test
void provideWords_returnsArrayWithCorrectSize() {
    RandomWordProvider provider = new RandomWordProvider();
    Word[] words = provider.provideWords(5);
    assertNotNull(words);
    assertEquals(5, words.length);
}
```

Analyse: Dieser Test prüft nur, ob das Array die richtige Größe hat, aber nicht, ob die Werte sinnvoll sind, ob Fehlerfälle (z.B. negative Größe) behandelt werden oder ob das Verhalten bei Grenzwerten korrekt ist.

Begründung: Das ist nicht thourough, weil wichtige Randfälle und Fehlerfälle fehlen. Ein gründlicher Test würde auch negative oder sehr große Werte, leere Arrays und Ausnahmen abdecken.

5.2.3 Professional

Professional bedeutet, dass die Tests nach Best Practices geschrieben sind:
- Sie sind klar strukturiert (Arrange-Act-Assert)
- nutzen sprechende Namen
- liefern bei Fehlern verständliche Ausgaben

POSITIV-BEISPIEL

Test:

```
@Test
void access_returnsMissOnFirstAccess_andHitOnSecond() {
    Address address = new Address(0b0000_0000);
    CacheAccessResult missResult = cache.access(address);
    assertFalse(missResult.isHit());
    CacheAccessResult hitResult = cache.access(address);
    assertTrue(hitResult.isHit());
}
```

Analyse: Der Testname beschreibt genau, was geprüft wird. Die Schritte sind klar nach AAA (Arrange-Act-Assert) strukturiert. Es wird sowohl der Miss als auch der Hitt geprüft.

Begründung: Das ist professionell, weil der Test leicht verständlich, wartbar und aussagekräftig ist. Fehler lassen sich schnell zuordnen.

2. Positiv-Beispiel

Test:

```
@Test
void recordAccess_incrementsAccessAndHitOrMiss() {
    stats.recordAccess(true);
    stats.recordAccess(false);
    stats.recordAccess(true);

    assertEquals(3, stats.getAccessCount());
    assertEquals(2, stats.getHitCount());
    assertEquals(1, stats.getMissCount());
    assertEquals(2.0/3.0, stats.getHitRate());
    assertEquals(1.0/3.0, stats.getMissRate());
}
```

Analyse:
- Der Testname beschreibt präzise, was geprüft wird.
- Die Methode ist nach Arrange-Act-Assert aufgebaut.
- Es werden mehrere Aspekte geprüft (Zugriffszähler, Treffer, Misses, Raten).
- Die Assertions sind klar und verständlich.

Begründung: Das ist professionell, weil der Test umfassend, aussagekräftig und leicht wartbar ist. Fehler lassen sich direkt nachvollziehen, und die Testlogik ist für jeden Entwickler verständlich.

5.3 Code Coverage

Die Code Coverage-Analyse zeigt, wie viel Prozent deines Codes durch Unit-Tests abgedeckt sind. Gesamtüberblick
- Klassenabdeckung: 92,6 % (25 von 27 Klassen)
- Methodenabdeckung: 84,8 % (89 von 105 Methoden)

- Zeilenabdeckung: 81,2 % (267 von 329 Codezeilen)

Fast alle Klassen und Methoden werden von Tests erreicht, und mehr als 80 % aller Codezeilen werden tatsächlich ausgeführt, wenn die Tests laufen.

Analyse:

- Domain-Modelle und Policies: Klassen wie Cache, CacheSet, CacheLine, CacheStatistics sowie die Ersetzungsstrategien (FIFO, LRU, LFU, Random) haben eine sehr hohe Abdeckung (meist über 95 %). Das ist sinnvoll, weil hier die Kernlogik steckt.
- Provider, Repository, Konfiguration: Auch diese Bereiche sind zu 100 % abgedeckt. Das zeigt, dass alle Wege zur Datenbereitstellung und -speicherung getestet werden.
- Application Layer (CacheService): 80 % der Methoden und 71,4 % der Zeilen sind abgedeckt. Die wichtigsten Anwendungsfälle werden also geprüft, einige Randfälle könnten noch ergänzt werden.
- Cache-Typen (DirectMappedCache, FullyAssociativeCache, NWayAssociativeCache): Hier ist die Abdeckung etwas niedriger (60 % Methoden, 42,9 % Zeilen). Das liegt daran, dass diese Klassen oft nur Konstruktoren mit spezifischen Parametern bereitstellen und die eigentliche Logik in der Basisklasse liegt.
- Presentation Layer (ConsoleUI): Die Abdeckung ist hier mit 18,6 % (Zeilen) am niedrigsten. Das ist typisch, weil UI-Logik schwer automatisiert zu testen ist. Die Kernlogik bleibt davon aber unberührt.
- Hilfsklassen (util): Liegen bei ca. 72 % Zeilenabdeckung. Hier könnten noch mehr Tests für Spezialfälle ergänzt werden.

5.4 Fakes und Mocks

Mocks und Fakes ermöglichen es, einzelne Komponenten isoliert zu testen, Interaktionen gezielt zu prüfen und Seiteneffekte auszuschließen

5.4.1 Beispiel 1: Mock-Objekt für `CacheRepository` im `CacheServiceTest`

Einsatz und Begründung:

- Im Test für die Klasse `CacheService` wird das Interface `CacheRepository` gemockt. So kann der Service isoliert getestet werden, ohne dass eine echte Repository-Implementierung (z.B. mit echter Speicherung) benötigt wird.
- Dadurch lassen sich gezielt Rückgabewerte und Verhalten des Repositories vorgeben und überprüfen, ob der Service korrekt mit dem Repository interagiert (z.B. ob `save()` oder `findById()` wie erwartet aufgerufen werden).

Test:

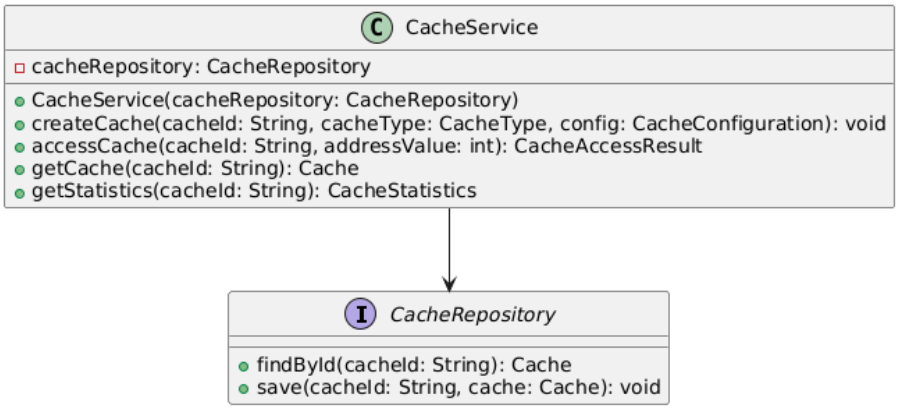
```
@Mock
private CacheRepository cacheRepository;

@InjectMocks
private CacheService cacheService;

@Test
void createCache_savesCacheInRepository() {
    CacheConfiguration config = mock(CacheConfiguration.class);
    when(config.getSetCount()).thenReturn(4);
    when(config.getAssociativity()).thenReturn(2);
    when(config.getWordsPerLine()).thenReturn(4);
    when(config.getReplacementStrategy())
      .thenReturn(mock(ReplacementStrategy.class));
    when(config.getWordProvider()).thenReturn(mock(WordProvider.class));
    cacheService.createCache("testCache", CacheType.N_WAY, config);
    verify(cacheRepository, times(1)).save(eq("testCache"),
    any(Cache.class));

}
```

UML-Diagramm:



Analyse:

- Das Mock-Objekt für `CacheRepository` sorgt dafür, dass ausschließlich die Logik von `CacheService` getestet wird.
- Interaktionen können gezielt überprüft werden (z.B. ob `save()` wirklich aufgerufen wurde).
- Änderungen am Repository (z.B. Datenbankanbindung) beeinflussen die Tests nicht.

5.4.2 Beispiel 2: Mocks für Service und Renderer in `ConsoleUITest`

Einsatz und Begründung:

- Im Test für die Klasse `ConsoleUI` werden mehrere Abhängigkeiten gemockt: `CacheService`, `CacheStateRenderer` und `SimulationStepPrinter`.
- Dadurch kann das Verhalten der Benutzeroberfläche isoliert getestet werden, ohne dass echte Caches, Services oder Ausgaben erzeugt werden müssen.

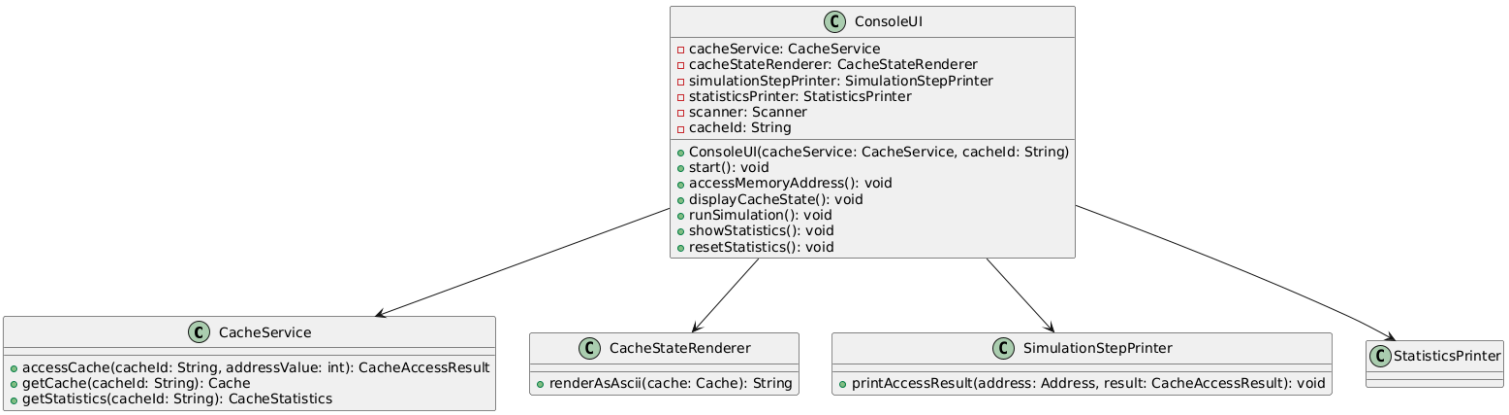
Test:

```java
private CacheService mockService = mock(CacheService.class);
private CacheStateRenderer mockRenderer = mock(CacheStateRenderer.class);
private SimulationStepPrinter mockPrinter =
mock(SimulationStepPrinter.class);

@BeforeEach
void setUp() {
    mockService = mock(CacheService.class);
    mockRenderer = mock(CacheStateRenderer.class);
    mockPrinter = mock(SimulationStepPrinter.class);
    ui = new ConsoleUI(mockService, "testCache");
}

@Test
void displayCacheState_printsCacheState() {
    Cache mockCache = mock(Cache.class);
    when(mockService.getCache("testCache")).thenReturn(mockCache);
        //...
      }
```

UML-Diagramm:



Analyse:

- Die Tests für `ConsoleUI` sind unabhängig von der echten Service- und Renderer-Logik.
- Es kann gezielt geprüft werden, ob die UI die richtigen Methoden aufruft und wie sie auf verschiedene Rückgabewerte reagiert.
- Fehler in der Service- oder Renderer-Implementierung beeinflussen die UI-Tests nicht.

6

Domain Driven Design

6.1 Ubiquitous Language

Die Ubiquitous Language stellt sicher, dass alle Beteiligten – Entwickler, Fachexperten und Stakeholder – dieselben Begriffe verwenden und ein gemeinsames Verständnis der Domäne haben.

Bezeichnung	**Bedeutung**	**Begründung**
„Cache Hit“	„Ein erfolgreicher Zugriff auf den Cache, bei dem die angeforderten Daten bereits im Cache vorhanden sind.“	„Ist sowohl in der Informatik als auch in der Fachdomäne des Cache-Managements gängig. Er wird durchgängig im Code (`CacheAccessResult.isHit()`) und in der Console verwendet und ist für alle Beteiligten eindeutig verständlich.“
„Cache Miss“	„Ein Zugriff auf den Cache, bei dem die angeforderten Daten nicht im Cache vorhanden sind und aus dem Hauptspeicher geladen werden müssen.“	„Zusätzlich zum Cache Hit ist dies ein fundamentaler Begriff der Cache-Domäne. Er wird konsistent in der gesamten Anwendung verwendet und ist sowohl für

Bezeichnung	Bedeutung	Begründung
		Entwickler als auch für Fachexperten sofort verständlich.“
„Replacement Strategy“	„Eine Strategie zur Bestimmung, welche Cache-Line ersetzt wird, wenn der Cache voll ist (z.B. FIFO, LRU, ...).“	„Dieser Begriff stammt direkt aus der Fachdomäne und wird sowohl im Code (`ReplacementStrategy`) als auch in fachlichen Diskussionen verwendet. Es beschreibt ein zentrales Konzept, das für das Verständnis des Cache-Verhaltens essentiell ist.“
„Set-Associative“	„Eine Cache-Architektur, bei der der Cache in Sets unterteilt ist und jedes Set mehrere Cache-Lines enthalten kann.“	„Dieser Begriff ist spezifisch für die Cache-Architektur und wird sowohl in technischen Spezifikationen als auch im Code (`NWayAssociativeCache`) verwendet.“

6.2 Entities

Entities sind Objekte mit einer dauerhaften Identität, die sich auch dann nicht ändert, wenn sich ihre Attribute ändern. Im Cache-Simulator gibt es keine klassische Entity, weil kein Objekt eine fachliche Identität über die Lebensdauer der Anwendung hinweg besitzt.

Begründung:
- Ein Cache, CacheSet oder CacheLine existiert nur für die Laufzeit und hat keine fachliche ID (wie z.B. eine Kundennummer).
- Die Identität ergibt sich rein aus der Position im Speicher (z.B. Array-Index), nicht aus einer fachlichen Bedeutung.
- Es gibt keine Geschäftsobjekte, die über längere Zeiträume hinweg eindeutig identifizierbar sein müssen.

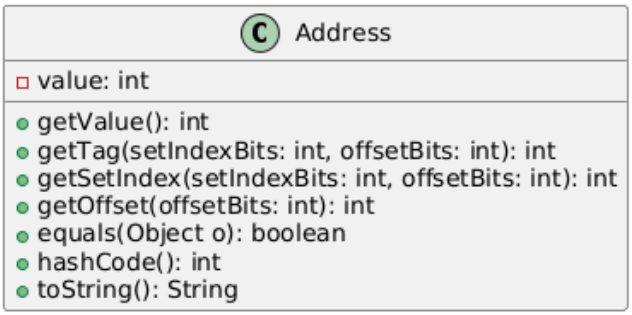
6.3 Value Objects

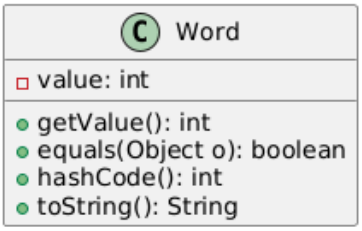
Value Objects sind Objekte, deren Gleichheit sich ausschließlich über ihre Werte definiert, nicht über eine Identität. SIe sind typischerweise immutable und können beliebig oft neu erzeugt oder ersetzt werden.

Beispiel aus dem Code:
- `Address`
- `Word`
- `CacheAccessResult`

Beschreibung und Begründung:
- Zwei `Address`-Objekte mit demselben Wert sind gleich, unabhängig davon, wann oder wie sie erzeugt wurden.
- `Word` ist ein Wrapper um einen Integer-Wert – Gleichheit basiert nur auf dem Wert.
- `CacheAccessResult` ist ein reines Ergebnisobjekt, das nur von den Feldern hit und word abhängt.

UML (Address und Word):





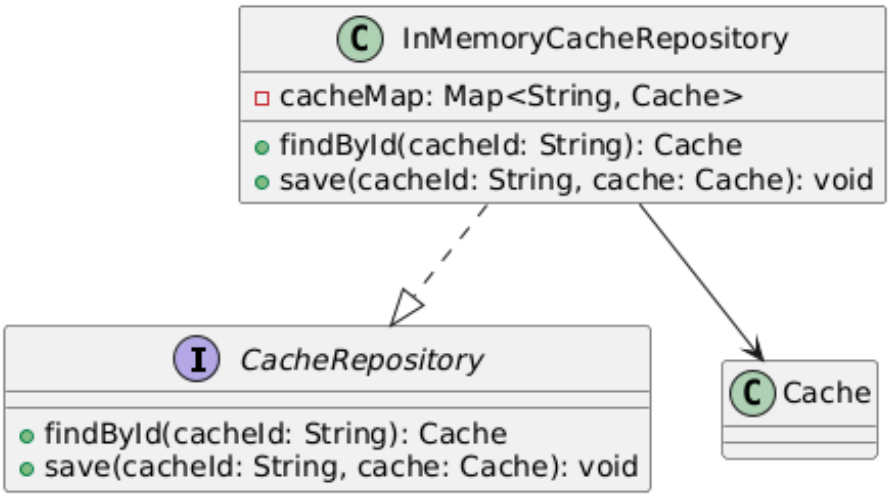
6.4 Repositories

Ein Repository ist ein Baustein aus DDD, der als Sammlung für Entities oder Aggregate dient und den Zugriff auf diese kapselt. Im Cache-Simulator gibt es ein Repository für die Verwaltung von Cache-Objekten.

Beispiel aus dem Code:
- Interface: `CacheRepository`
- Implementierung: `InMemoryCacheRepository`

Beschreibung und Begründung:
- Das Repository abstrahiert die Speicherung und das Auffinden von Caches über eine ID.
- Die Implementierung ist technisch (In-Memory), aber das Muster ist identisch zu typischen DDD-Repositories.
- Auch wenn die gespeicherten Objekte keine klassischen Entities sind, ist das Repository sinnvoll, um die Verwaltung und den Zugriff zu kapseln.

UML:



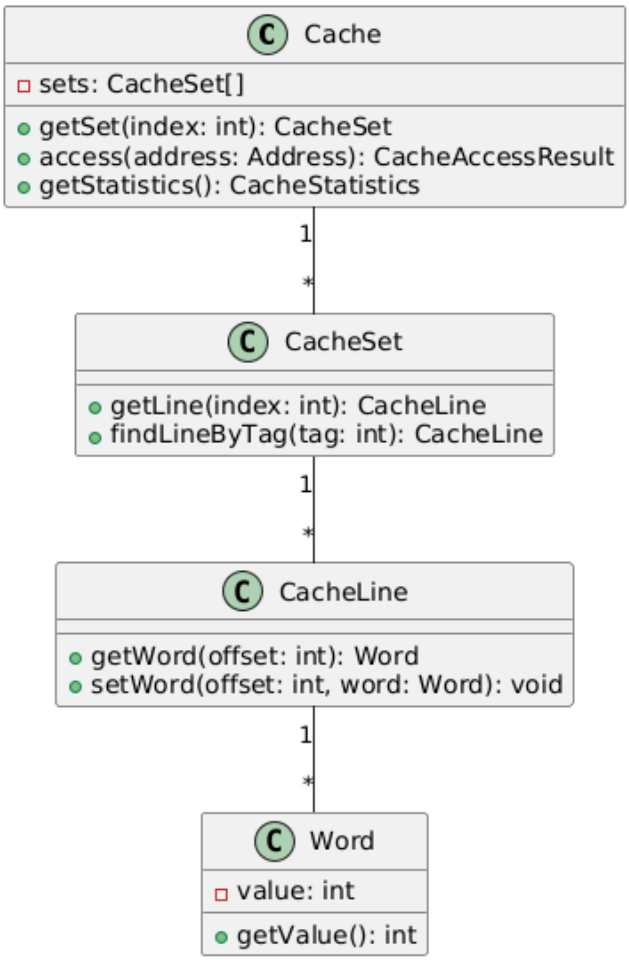
6.5 Aggregates

Ein Aggregate ist im DDD ein Cluster aus Entities und Value Objects, das als eine Konsistenzeinheit betrachtet wird. Das Aggregate hat einen Root, über dem alle Zugriffe laufen.

Beispiel aus dem Code:
- Das Aggregate im Cache-Simulator ist das `Cache`-Objekt mit seinen untergeordneten `CacheSet`- und `CacheLine`-Objekten.

Beschreibung und Begründung:
- `Cache` ist die Aggregate Root: Alle Zugriffe auf Sets und Lines laufen über das Cache-Objekt.
- Konsistenzregeln (z.B. wie Daten geschrieben oder gelesen werden) werden zentral im Cache geprüft.
- Die untergeordneten Objekte (`CacheSet`, `CacheLine`, `Word`) sind ohne das Cache-Objekt nicht sinnvoll nutzbar.

UML:


Cache
sets: CacheSet[]
getSet(index: int): CacheSet
access(address: Address): CacheAccessResult
getStatistics(): CacheStatistics
1
*
CacheSet
getLine(index: int): CacheLine
findLineByTag(tag: int): CacheLine
1
*
CacheLine
getWord(offset: int): Word
setWord(offset: int, word: Word): void
1
*
Word
value: int
getValue(): int

7

Refactoring

7.1 Code Smells

7.1.1 Code Smell 1: Switch Statements

Gefunden in: `CacheFactory.java`

Problemanalyse:

- Switch-Statements sind ein klassischer Code Smell in objektorientierten Sprachen
- Bei neuen Cache-Typen muss die Factory-Klasse modifiziert werden
- Verletzt das Open-Closed Principle - nicht offen für Erweiterung, nicht geschlossen für Modifikation
- Die Methode hat mehrere Verantwortlichkeiten und wird bei jeder Erweiterung geändert

Code:

```
public static Cache createCache(CacheType type, CacheConfiguration
config) {
    if (type == null) {
        throw new IllegalArgumentException("Cache type is not allowed to
be null");
    }
    return switch (type) {
        case DIRECT_MAPPED -> new
DirectMappedCache(config.getSetCount(),
                              config.getWordsPerLine(),
config.getReplacementStrategy(),
                              config.getWordProvider());
        case FULLY_ASSOCIATIVE -> new
FullyAssociativeCache(config.getAssociativity(),
                                   config.getWordsPerLine(),
config.getReplacementStrategy(),
                                   config.getWordProvider());
        case N_WAY -> new NWayAssociativeCache(config.getSetCount(),
                         config.getAssociativity(),
config.getWordsPerLine(),
                         config.getReplacementStrategy(),
config.getWordProvider());
        default -> throw new IllegalArgumentException("Unsupported cache
type");
    };
}
```

Lösungsweg -> Abstract Factory Pattern

Interface für Cache-Type-Factories:

```
public interface CacheTypeFactory {
    Cache createCache(CacheConfiguration config);
}
```

Konkrete Factory-Implementierungen:

```
public class DirectMappedCacheFactory implements CacheTypeFactory {
    @Override
    public Cache createCache(CacheConfiguration config) {
        return new DirectMappedCache(config.getSetCount(),
                                     config.getWordsPerLine(),
                                     config.getReplacementStrategy(),
                                     config.getWordProvider());
    }
}

public class FullyAssociativeCacheFactory implements CacheTypeFactory {
    @Override
    public Cache createCache(CacheConfiguration config) {
        return new FullyAssociativeCache(config.getAssociativity(),
                                         config.getWordsPerLine(),
                                         config.getReplacementStrategy(),
                                         config.getWordProvider());
    }
}
```

Refaktorierte CacheFactory:

```
public class CacheFactory {
    private static final Map<CacheType, CacheTypeFactory> factories =
Map.of(
        CacheType.DIRECT_MAPPED, new DirectMappedCacheFactory(),
        CacheType.FULLY_ASSOCIATIVE, new FullyAssociativeCacheFactory(),
        CacheType.N_WAY, new NWayAssociativeCacheFactory()
    );

    public static Cache createCache(CacheType type, CacheConfiguration
config) {
        CacheTypeFactory factory = factories.get(type);
        if (factory == null) { throw new IllegalArgumentException
        ("Unsupported cache type: " + type);
        }
        return factory.createCache(config);
    }
}
```

Nach der Lösung:

- Neue Cache-Typen können hinzugefügt werden, ohne die bestehende Cache-Factory zu ändern
- Jede Factory hat nur eine einzige Verantwortung (Single Responsibility Principle)
- Bessere Erweiterbarkeit und Wartbarkeit
- Elimination des Switch-Statement Code Smells

7.1.2 Code Smell 2: Duplicated Code

Gefunden in: Test-Klassen und `ConsoleUI.java` Aktueller Code (Problem):

```
// In DirectMappedCacheTest.java:
@BeforeEach
void setUp() {
    cache = new DirectMappedCache(4, 2, new FIFOStrategy(), new StaticWordProvider(11));
}

// In FullyAssociativeCacheTest.java:
@BeforeEach
void setUp() {
    cache = new FullyAssociativeCache(3, 2, new FIFOStrategy(), new StaticWordProvider(5));
}

// In NWayAssociativeCacheTest.java:
@BeforeEach
void setUp() {
    cache = new NWayAssociativeCache(2, 2, 2, new FIFOStrategy(), new StaticWordProvider(7));
}
```

```java
// In ConsoleUI.java - Menü-Ausgabe in der start() Methode:
System.out.println("\nPlease choose an option:");
System.out.println("1. Access memory address");
System.out.println("2. Display cache state");
System.out.println("3. Run simulation");
System.out.println("4. Show statistics");
System.out.println("5. Reset statistics");
System.out.println("6. Exit");
```

Problemanalyse:

- Duplicated Code ist der häufigste und wichtigste Code Smell
- Gleiche oder sehr ähnliche Logik zur Cache-Erstellung in mehreren Test-Klassen
- Menü-Ausgabe wird bei jeder Anzeige wiederholt
- Verletzt das DRY-Prinzip (Don't Repeat Yourself)
- Erhöht den Wartungsaufwand bei Änderungen

Lösungsweg -> Extract Method & Helper Classes

```java
// Test Helper Klasse für gemeinsame Cache-Erstellung
public class CacheTestHelper {
    public static Cache createTestCache(CacheType type, int sets, int
associativity,
                                         int wordsPerLine, int staticValue)
{
        CacheConfiguration config = new CacheConfiguration(
            sets, associativity, wordsPerLine,
            new FIFOStrategy(), new StaticWordProvider(staticValue)
        );
        return CacheFactory.createCache(type, config);
    }
}
```

Refaktorierte Test-Klassen:

```
// DirectMappedCacheTest.java:
@BeforeEach
void setUp() {
    cache = (DirectMappedCache) CacheTestHelper.createTestCache(
        CacheType.DIRECT_MAPPED, 4, 1, 2, 11);
}

// FullyAssociativeCacheTest.java:
@BeforeEach
void setUp() {
    cache = (FullyAssociativeCache) CacheTestHelper.createTestCache(
        CacheType.FULLY_ASSOCIATIVE, 1, 3, 2, 5);
}
```

Menu Helper für ConsoleUI:

```
public class MenuHelper {
    private static final String[] MENU_OPTIONS = {
        "1. Access memory address",
        "2. Display cache state",
        "3. Run simulation",
        "4. Show statistics",
        "5. Reset statistics",
        "6. Exit"
    };

    public static void displayMenu() {
        System.out.println("\nPlease choose an option:");
        for (String option : MENU_OPTIONS) {
            System.out.println(option);
        }
    }
}
```

Refaktorierte ConsoleUI:

```
public void start() {
    System.out.println("Welcome to my Cache Simulator");
    while (true) {
        MenuHelper.displayMenu();
        int choice = scanner.nextInt();
        // ... rest of logic
    }
}
```

Nach der Lösung:
- Zentralisierte Cache-Erstellung reduziert Code-Duplikation erheblich
- Änderungen am Menü müssen nur an einer Stelle vorgenommen werden
- Tests werden konsistenter und wartbarer
- Befolgt das DRY-Prinzip und reduziert Wartungsaufwand
- Bessere Lesbarkeit und Struktur des Codes

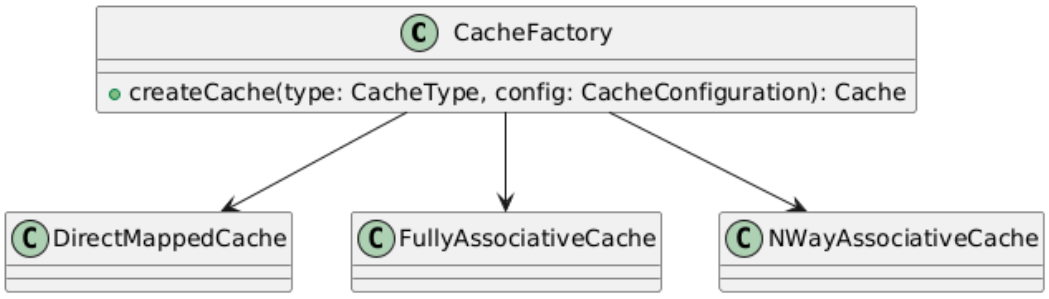
7.2 Refactoring

7.2.1 Replace Switch Statement with Polymorphism

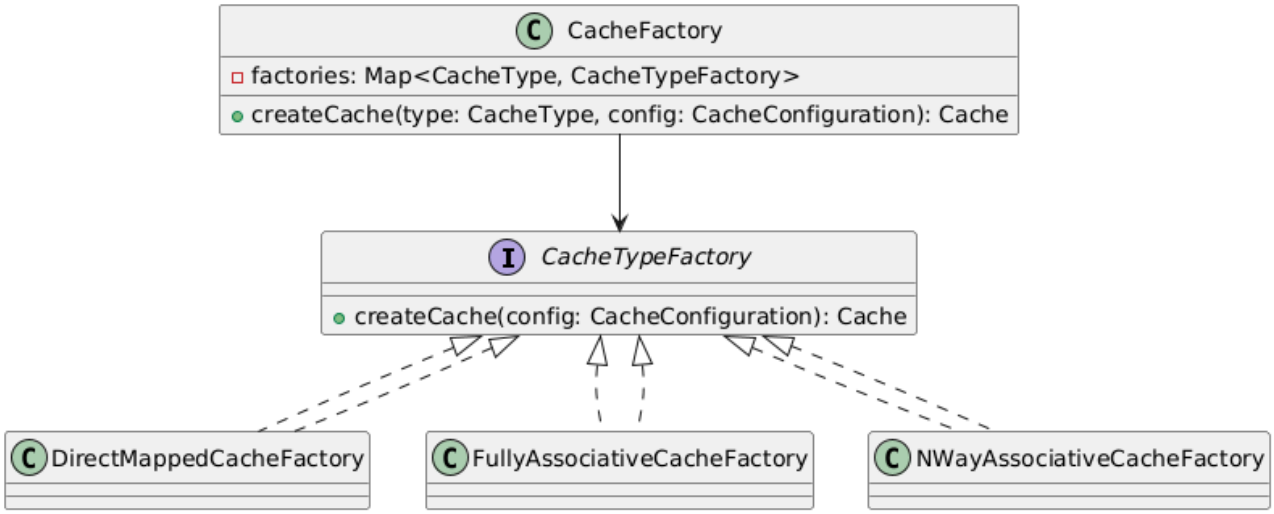
Commit: 7fec66d

Begründung: Im ursprünglichen Code der CacheFactory wurde ein Switch-Statement verwendet, um je nach CacheType die passende Cache-Implementierung zu erzeugen. Das ist ein klassischer Code Smell (Switch Statement), weil bei jeder Erweiterung der Cache-Typen die Factory angepasst werden muss. Das verletzt das Open-Closed Principle und erschwert die Wartung.

Vorher:



Nachher:



Ergebnis:

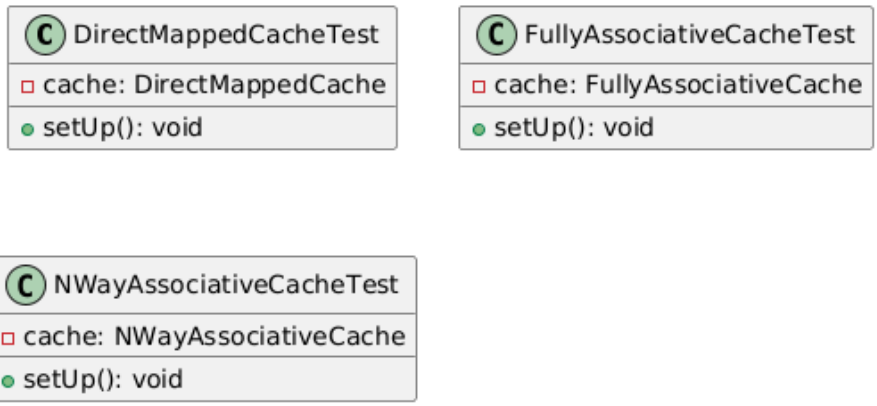
- Kein Switch-Statement mehr, sondern Polymorphie über Factories.
- Neue Cache-Typen können hinzugefügt werden, ohne die zentrale Factory zu ändern.
- Bessere Erweiterbarkeit und Wartbarkeit.

7.2.2 Extract Method (DRY – Duplicated Code entfernen)

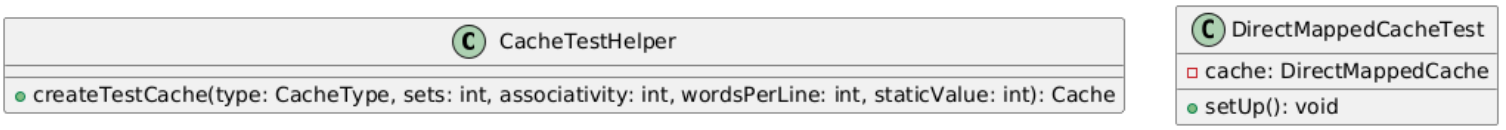
Commit: 48e127c

Begründung: In den Testklassen für verschiedene Cache-Typen (z.B. DirectMappedCacheTest, FullyAssociativeCacheTest, NWayAssociativeCacheTest) wurde die Erzeugung von Test-Caches mehrfach fast identisch implementiert. Das ist ein klassischer Duplicated Code Smell. Durch das Extrahieren in eine Hilfsmethode wird der Code wartbarer und Änderungen müssen nur noch an einer Stelle erfolgen.

Vorher:



Nachher:



Ergebnis:

- Die Cache-Erstellung ist zentralisiert.
- Kein Duplicated Code mehr in den Testklassen.
- Änderungen am Test-Setup müssen nur noch an einer Stelle gemacht werden.

8

Entwurfsmuster

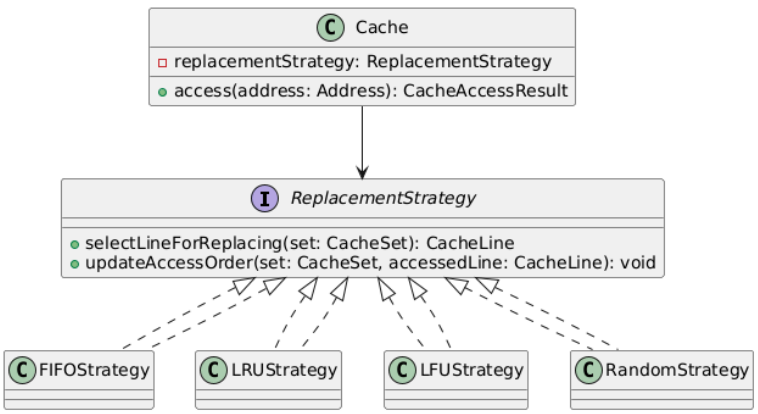
Im Projekt werden verschiedene Entwurfsmuster eingesetzt, um die Struktur und Erweiterbarkeit des Codes zu verbessern.

8.1 Entwurfsmuster 1: Strategy Pattern

Einsatz im Projekt: Das Strategy Pattern wird für die Austauschbarkeit der Cache-Ersetzungsstrategien genutzt. Die zentrale Schnittstelle ist `ReplacementStrategy`, die von mehreren Klassen wie `FIFOStrategy`, `LRUStrategy`, `LFUStrategy` und `RandomStrategy` implementiert wird.

Begründung:
- Die Ersetzungsstrategie kann zur Laufzeit ausgewählt und gewechselt werden.
- Neue Strategien lassen sich hinzufügen, ohne den Cache-Code zu ändern (OCP).
- Der Cache bleibt unabhängig von der konkreten Strategie.

UML-Diagramm:



8.2 Entwurfsmuster 2: Factory Pattern

Einsatz im Projekt: Das Factory Pattern wird in der Klasse `CacheFactory` verwendet, um verschiedene Cache-Typen (`DirectMappedCache`, `FullyAssociativeCache`, `NWayAssociativeCache`) zu erzeugen. Die Factory übernimmt die Instanziierung und verbirgt die konkrete Implementierung vor dem Aufrufer.

Begründung:
- Die Erzeugung von Cache-Objekten ist gekapselt und zentralisiert.
- Neue Cache-Typen können leicht hinzugefügt werden, indem die Factory erweitert wird.
- Der Code, der einen Cache benötigt, muss keine Details über die Konstruktion kennen.

UML-Diagramm: