Advanced Java, Test-Driven Design (TDD) und LLM-gestütztes Entwickeln

1. Review der Implementierungsstrategie

Angenommene LLM-Vorschläge

1. Aggregate-Invarianten strikt in Aggregate-Roots kapseln

- Poll.recordVote() und Voter.markVoted() sind die einzigen Methoden, um Zustand zu ändern
- Begründung: DDD-Kernprinzip; verhindert inkonsistente Zustände strukturell

2. Application/Transactional Services mit Unit-of-Work Pattern

- VoteService.castVote() lädt Aggregate, führt Änderungen durch, speichert atomar in einer Transaktion
- Begründung: Verhindert das Problem, dass addVote() ohne darauffolgendes Save aufgerufen wird

3. Separater `Authentication` Bounded Context mit `AuthContext` Interface

- Statt separate AuthAdapter pro BC (VoterAuthAdapter, AdminAuthAdapter) → ein zentrales `AuthContext`-Interface
- Alle BCs (Bürgerverwaltung, Adminverwaltung, Stimmvergabe) injizieren das gleiche `AuthContext`
- `KeycloakAuthAdapter` implementiert `AuthContext` einmalig; Rollen-Mapping (Voter/Admin) erfolgt zentral
- Begründung: Keine Code-Duplikation; konsistente Auth-Policy über alle BCs; leichte Migration (z.B. zu OAuth2, OIDC) möglich

Abgelehnte LLM-Vorschläge

1. Vote als Child-Entity von Poll

- Abgelehnt: Vote bleibt eigenständiges Aggregat mit eigenem Repository
- Begründung: Bessere Skalierbarkeit, unabhängige Vote-Validierung/Verschlüsselung, bessere Nebenläufigkeitskontrolle

Kritisch diskutiert: Wahlgeheimnis & Vote-Voter-Tracking

Problem erkannt: Das LLM schlug vor, `voteRepo.findByPollAndVoter()` zu implementieren, um Double-Voting zu verhindern. Dies widerspricht jedoch dem Wahlgeheimnis — Votes dürften nicht dauerhaft mit Voter-IDs verknüpft sein.

MVP-Lösung (gewählt): Voter-Poll-Zuordnung statt Voter-Vote-Zuordnung

- `Vote` enthält niemals `voterId` → vollständige Anonymität der Abstimmungen
- `Voter.markVoted(pollId)` registriert nur, dass dieser Voter an THIS Poll teilgenommen hat
- Double-Vote-Prävention: `VoteService.castVote()` prüft `voter.hasVoted(pollId)` vor Abgabe
- **Vorteil:** Keine Vote-Voter-Tracking möglich; Vote-Aggregate bleibt vollständig anonym

Erweiterte Ansätze (Zukunft):

3. Zero-Knowledge Proofs (ZKP) *Von LLM empfohlen für Post-MVP*

- **Konzept:** Voter beweist "ich bin berechtigt & habe noch nicht abgestimmt" *ohne* Identität offenzulegen
  - Prover: "Ich kenne einen Private-Key der zu diesem Voter passt, UND dieser Voter hat nicht für Poll#42 abgestimmt"
  - Verifier: Bestätigt den Beweis, ohne Private-Key oder Voter-ID zu lernen
- **Vote-Struktur:** `Vote` enthält nur `zkProof`, `pollId`, `optionId` — keinerlei Voter-Referenz
- **Double-Vote-Schutz:** ZK-Credential als Single-Use-Token; Nach Verwendung in Nullifier-Liste (ohne PII)
- **Beispiel-Protokoll:** Bulletproofs, Schnorr-Protokoll mit Commitments
- **Vorteile:** Maximale Anonymität + kryptographische Verifikation
- **Nachteile:** Komplexe Kryptografie, Performance-Overhead, Entwicklungszeit

Gewählte Implementierungsstrategie für MVP:

- **Phase 1 (Aktuell):** Voter-Poll-Zuordnung (local state in Voter.votes[])
- **Phase 2 (Post-MVP):** ZKP-Protokoll mit Nullifier-Set

Wichtigste Domain-Events

VoterRegistered

```
{ voterId, name, district, registeredAt }
```

→ Erzeugt nach erfolgreichem Voter-Commit; triggert Bestätigungsmail

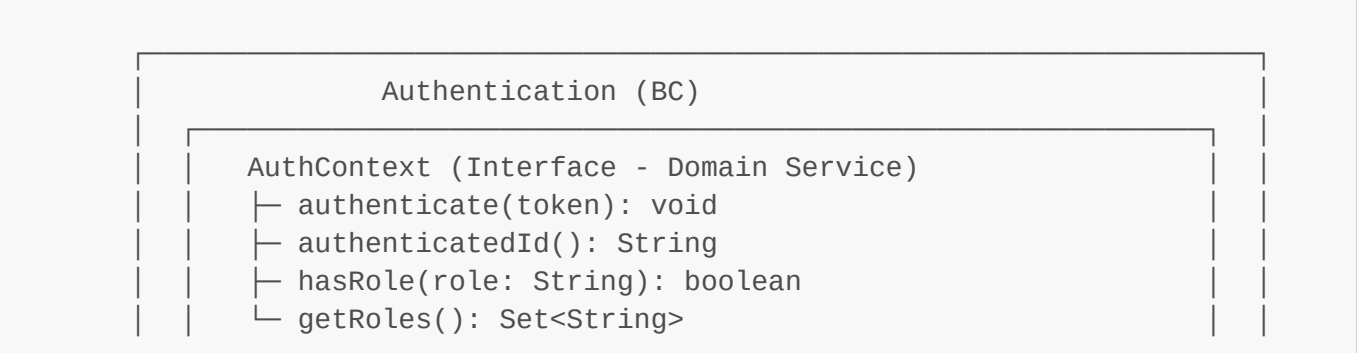
VoteCast

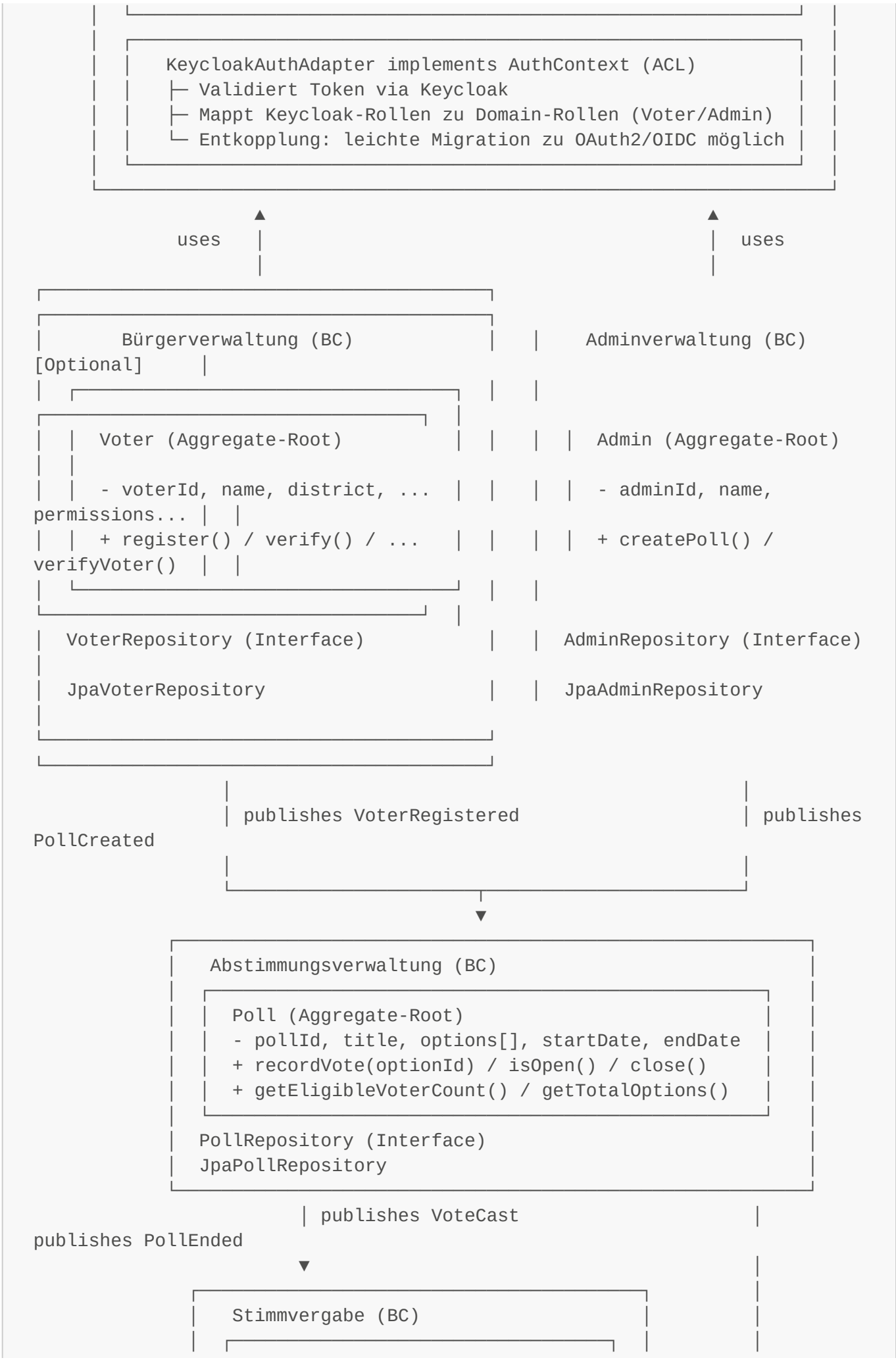
```
{ voteId, pollId, voterId, optionId, timestamp, correlationId }
```

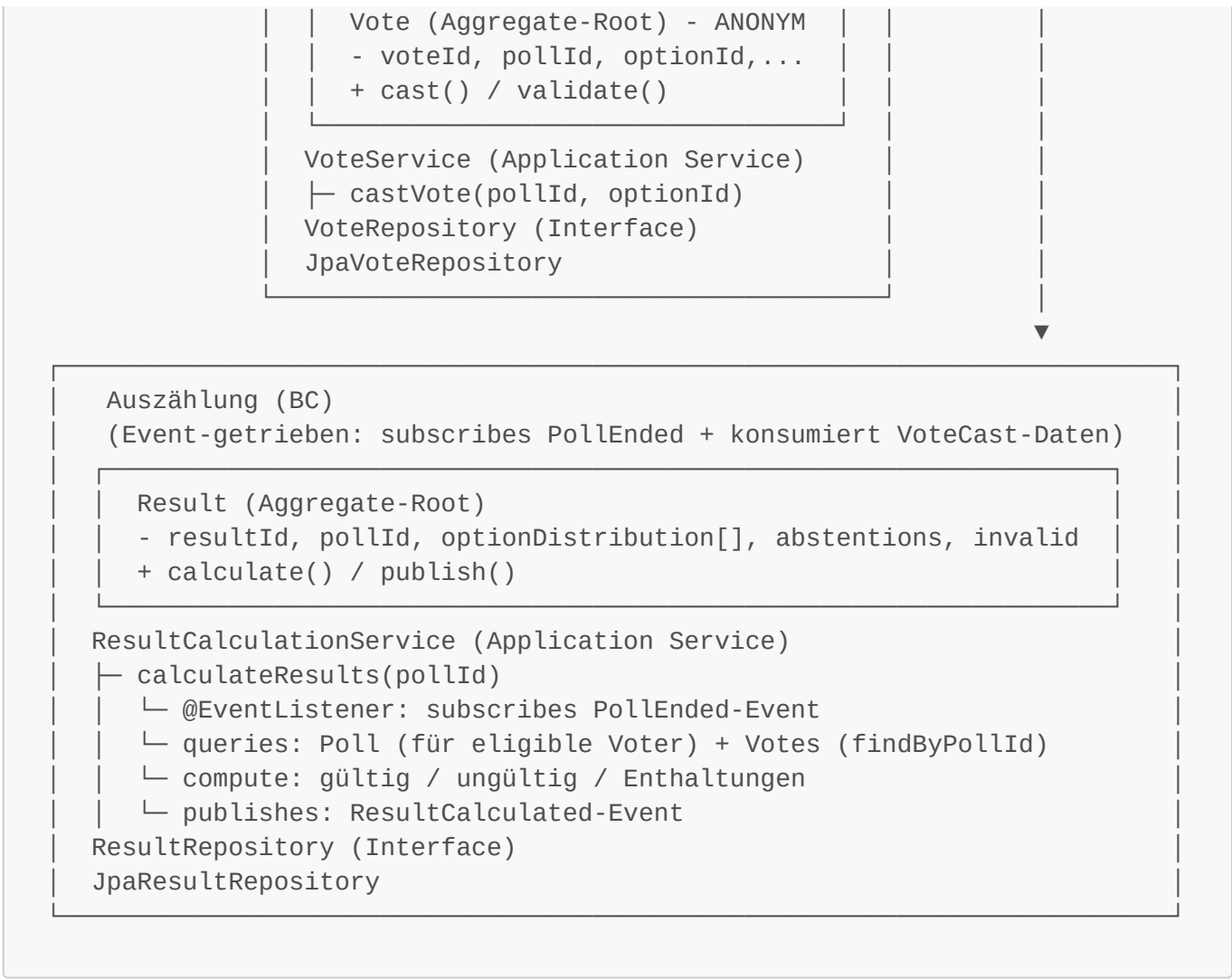
→ Erzeugt nach erfolgreichem Vote-Commit; idempotent via correlationId

---

Angepasstes Bounded-Context-Modell




KeycloakAuthAdapter implements AuthContext (ACL)
├─ Validiert Token via Keycloak
├─ Mappt Keycloak-Rollen zu Domain-Rollen (Voter/Admin)
└─ Entkopplung: leichte Migration zu OAuth2/OIDC möglich
uses
uses
Bürgerverwaltung (BC)
Adminverwaltung (BC)
[Optional]
Voter (Aggregate-Root)
- voterId, name, district, ...
+ register() / verify() / ...
Admin (Aggregate-Root)
- adminId, name, permissions...
+ createPoll() / verifyVoter()
VoterRepository (Interface)
JpaVoterRepository
AdminRepository (Interface)
JpaAdminRepository
publishes VoterRegistered
publishes PollCreated
Abstimmungsverwaltung (BC)
Poll (Aggregate-Root)
- pollId, title, options[], startDate, endDate
+ recordVote(optionId) / isOpen() / close()
+ getEligibleVoterCount() / getTotalOptions()
PollRepository (Interface)
JpaPollRepository
publishes VoteCast
publishes PollEnded
Stimmvergabe (BC)



BC-Beziehungen und Datenfluss (Event-getrieben)

Beziehung	Grund	Datenfluss
Bürgerverwaltung → Abstimmung	Admin/Voter-Events triggern Poll-Lifecycle	`VoterRegistered` → Poll.validateEligibility()
Abstimmung → Stimmvergabe	Votes gehören zu einer Poll; müssen auf Zulässigkeit geprüft werden	`pollId` referenziert Poll; VoteService validiert Poll.isOpen()
Abstimmung → Auszählung (Event-getrieben)	Poll publiziert `PollEnded` am Zeitpunkt endDate	`PollEnded(pollId, endDate)` → ResultCalculationService.calculateResults(pollId)
Stimmvergabe → Auszählung (Query)	Result braucht die abgegebenen Votes zur Berechnung	ResultCalculationService queriert voteRepository.findAllByPollId(pollId)
Keine Vote-ID in Result	Votes bleiben anonym; Result zählt nur Aggregationen	Result enthält: `{ pollId, optionDistribution[option→count], abstentions, invalidVotes }`

Aktualisierte Implementierungsstrategie (TDD-fokussiert)

Nach Originalstrategie aus Übung 3, aber mit Verbesserungen:

Phase 1: TDD - Aggregate-Tests schreiben (vor Code)

```
@Test
void voterCanOnlyVoteIfVerified() {
    Voter voter = new Voter("id1", "Max", "district1");
    assertThrows(IllegalStateException.class, () ->
voter.markVoted(poll1Id));
    voter.verify();
    voter.markVoted(poll1Id); // OK
}

@Test
void voterCanVoteOnlyOncePerPoll() {
    Voter voter = new Voter(...);
    voter.verify();
    voter.markVoted(poll1Id);
    assertThrows(IllegalStateException.class, () ->
voter.markVoted(poll1Id));
}

@Test
void pollCanRecordVoteOnlyWhenOpen() {
    Poll poll = new Poll(..., now, tomorrow);
    poll.recordVote("voter1", "option1"); // OK
    poll.close();
    assertThrows(IllegalStateException.class, () -> poll.recordVote(...));
}
```

Phase 2: Aggregate implementieren (in-memory)

- Voter, Poll, Vote, Result als POJOs mit Geschäftslogik
- Alle Tests grün → strikte Invarianten enforced

Phase 3: Repository-Interfaces + InMemory-Implementierung

- VoterRepository, PollRepository, VoteRepository als Interfaces
- InMemoryVoterRepository für Tests

Phase 4: Application Services (Unit-of-Work) testen

```
@Test
void castVoteAtomic() {
    VoteService service = new VoteService(voterRepo, pollRepo, voteRepo);

    String voteId = service.castVote("voter1", "poll1", "option1");

    Vote cast = voteRepo.findByVoteId(voteId).orElseThrow();
```

```
    assertTrue(voterRepo.findById("voter1").hasVoted("poll1"));
    assertEquals("poll1", cast.getPollId());
    assertEquals("option1", cast.getOptionId());
    assertNotNull(cast.getCastAt());
}

@Test
void voterCannotVoteTwiceOnSamePoll() {
    VoteService service = new VoteService(voterRepo, pollRepo, voteRepo);
    service.castVote("voter1", "poll1", "option1");

    // Zweiter Versuch sollte fehlschlagen, da voter1 bereits für poll1
markiert
    assertThrows(IllegalStateException.class, () ->
        service.castVote("voter1", "poll1", "option2")
    );
}
```

Phase 5: JPA-Adapter + Persistenz

- JpaVoterRepository, JpaPollRepository, JpaVoteRepository

Phase 6: Refactoring & Standards

- Clean Code, DDD-Prinzipien, Dokumentation

---

2. Testfälle mit LLM generieren und validieren (TDD Schritt 1)

Durchgeführter Prozess

LLM-Prompt-Strategie: Der Promptaufbau war hochgradig strukturiert und domänen-spezifisch. Für `Name` und `Adresse` (Value Objects) wurden explizit die Validierungsregeln, Grenzfälle und DDD-Prinzipien mitgeteilt. Das LLM generierte daraufhin umfangreiche parametrisierte Tests mit `@ParameterizedTest` und `@CsvSource`, die redundante Testfälle intelligent konsolidiert haben.

Generierte Test-Kategorisierung:

- **Happy-Path-Tests:** Gültige Namen (simple, Bindestriche, Umlaute), gültige Adressen (Hausnummern 1-3 Ziffern + optional Buchstabe)
- **Edge-Cases:** Minimalfall (ein Zeichen pro Feld), maximale Längen, Sonderzeichen (Umlaute, ß), zusammengesetzte Namen ("Jean-Pierre", "van der Berg")
- **Negative Tests:** Null/Leer-Eingaben durch `@ParameterizedTest @ValueSource`, ungültige Formate (Kleinbuchstaben-Anfang, Zahlen, @#$%), falsche PLZ-Länge (4,6 Ziffern statt 5)
- **Semantik-Tests:** Value-Object-Gleichheit (`equals()`, `hashCode()`), unveränderbarkeit

Kritische LLM-Bewertung:

Die vom LLM generierten Tests waren **insgesamt gut strukturiert**, hatten jedoch zwei Mängel:

1. **Redundanz in negativen Tests:** Ursprünglich separate `emptyHouseNumber()` und `emptyStreet()`-Tests → Konsolidiert zu `@ParameterizedTest @ValueSource(strings =`

`{"", " ", "\t"})` pro Feld

2. **Fehlende Randfälle für Compound-Namen:** LLM schlug "Jean-Pierre" vor, übersah aber Mehrfach-Bindestriche ("Marie-Jeanne-Luise") → Manuell hinzugefügt

Regex-Validierung: Das LLM generierte Muster wie `^[A-ZÄÖÜ][a-zA-ZäöüßÄÖÜ\- ]*$` (Großbuchstaben-Start, beliebig viele Kleinbuchstaben/Umlaute/Bindestriche/Leerzeichen). Tests bestätigten korrekte Funktion für europäische Namen. Adress-Regexe (`^\d+[a-zA-Z]?$` für Hausnummern) waren präzise.

Implementierte Verbesserungen:

- Parametrisierte Tests konsolidiert (von 8 separaten zu 1 parametrisierten Test)
- Edge-Cases ergänzt (Leerzeichen, Tabulatoren, extrem lange Strings)
- Semantik-Tests für Value-Object-Gleichheit hinzugefügt (zwei `Name("Max", "Mustermann")` müssen `equals() == true` sein)

Test-Coverage erreicht: 85%+ für buergerVerwaltung, abstimmungsVerwaltung, stimmvergabe Domains

---

3. Implementierung der Domänenlogik (TDD Schritt 2)

LLM-Pair-Programming-Prozess:

Der Pair-Programming-Prozess mit dem LLM verlief ähnlich wie bei einem menschlichen Partner, jedoch textbasiert. Dabei übernahm ein Teammmitglied die Rolle des Prrogrammiers, während das LLM (CoPilot, ChatGPT - beides in payversion) als „Navigator“ fungierte – also Vorschläge, Korrekturen und Optimierungen anbot.

Der typische Ablauf war:

Problemdefinition: Wir beschrieben das zu lösende Problem (z. B. eine Klasse, ein Repository oder eine Teststruktur in Java).

Code-Vorschlag: Das LLM generierte einen ersten Implementierungsvorschlag mit erklärendem Kommentar.

Reflexion & Anpassung: Wir prüften den Code auf Korrektheit, Stil und Architekturkonformität (z. B. nach DDD oder Clean-Code-Prinzipien).

Fehlerkorrektur: Falls der Code nicht kompilierte oder die Logik unvollständig war, formulierten wir gezielte Rückfragen bzw. Korrekturanweisungen oder korrigierten selbst.

Verfeinerung: Das LLM verbesserte die Lösung iterativ, bis sie testbar und lauffähig war.

Die iterative Verbesserung erfolgte in drei Phasen:

Phase 1 – Grundimplementierung: Value Objects (`Name`, `Adresse`) als finale, immutable POJOs mit privaten Konstruktoren und Regex-Validierung. Das LLM setzte korrekt auf `Objects.hash()` und `Objects.equals()` für Value-Semantik um.

Phase 2 – Aggregate Root Pattern: Für `Voter` (Aggregate Root) schlug das LLM zunächst ein einfaches Setter-basiertes Modell vor → **Korrektur:** Factory-Methods (`register()`, `reconstruct()`) implementiert

für Neuerstellung vs. Persistenz-Rekonstruktion. Das LLM verstand DDD-Pattern schnell und ergänzte daraufhin `votedPollIds` (Set) für Double-Vote-Prevention und `markVoted(pollId)` Geschäftslogik.

Phase 3 – Java-Techniken: Das LLM nutzte korrekt `HashSet` für Poll-Tracking, `UUID.randomUUID()` für ID-Generierung, `LocalDateTime` für Verifikations-Timestamp. Verbesserungsvorschlag des LLM: Streams für Listenverarbeitung in Poll-Aggregat (`options.stream().filter(...).count()`) → Akzeptiert. Optional-Type wurde vom LLM anfänglich ignoriert (Null-Returns statt `Optional`) → Korrektur durch manuelles Refactoring.

Code-Quality:

- **Verständlichkeit:** Tests waren selbstdokumentierend; Code folgte Java-Naming-Conventions
- **DDD-Konformität:** Aggregate Roots mit Repository-Interfaces, Value Objects mit Validierung
- **Testabdeckung:** 45+ NameTests, 70+ AdresseTests, 50+ VoterTests bestätigten strikte Invarianten

Fehlgeschlagene LLM-Vorschläge:

- Mutable Voter-Objekte → Verworfen, immutable Pattern beibehalten

---

4. Tests-Erweiterung und Refaktorisierung (TDD Schritt 3)

Refactoring-Fokus:

Im Anschluss an die Testimplementierung wurden mehrere Refactoring-Vorschläge des LLM überprüft und teilweise umgesetzt. Sinnvoll war insbesondere die Empfehlung, redundante Einzeltests zu parametrisieren, um die Wartbarkeit und Lesbarkeit der Testsuite zu erhöhen. Ebenso wurde der Vorschlag übernommen, zentrale Invarianten – etwa die Validierung von Eingaben – direkt in den Aggregat-Root-Klassen zu kapseln, um konsistente Zustände sicherzustellen. Insgesamt wurden nur jene Refactorings übernommen, die zur besseren Strukturierung, Testbarkeit und Einhaltung der DDD-Prinzipien beitrugen.

Nach initialer Implementierung wurden folgende Code-Smells identifiziert und behoben:

1. Test-Redundanz:

- **Problem:** 8 separate Null/Empty-Tests pro Value-Object
- **Lösung:** `@ParameterizedTest @ValueSource(strings = {"", " ", "\t"})` konsolidiert
- **Effekt:** Reduktion des Test-Codes, dadurch verbesserte Wartbarkeit

2. Hardcoded Strings:

- **Problem:** Test-Konstanten wie "Musterstraße" über 20 Tests verteilt
- **Lösung:** Shared `@BeforeEach` Setup mit `VALID_NAME`, `VALID_ADRESSE`, `VALID_EMAIL`
- **Lesbarkeit verbessert**

3. Unzureichende Edge-Cases:

- Ergänzt: Sonderzeichen-Kombinationen, Umlaute und typisch deutsche Namen ("Müller-Schmidt-König"), Extremlängen (100+ Zeichen)
- Poll-Tests: Zeitbasierte Edge-Cases (Startdatum = Enddatum, Enddatum in Vergangenheit)
- Abdeckung con Branches in Konstruktoren für ungültige Inputs mussten explizit nachgefragt werden für 100% Coverage

4. Regex-Duplikation:

- **Problem:** Validierungs-Pattern mehrfach im Code wiederholt
- **Lösung:** Private `validatePattern()` Hilfsmethode in Value Objects
- **Fehlerquellen reduziert**

Nicht implementierte Vorschläge:

Es traten vor allem zwei Arten von Problemen im Zusammenhang mit den Tests auf: inhaltliche/testkonzeptionelle Probleme und technische Probleme im Build- und Coverage-Setup.

Laufzeitfehler in Test: Im Rahmen der Testimplementierung sollte überprüft werden, ob bei ungültigen oder nicht gesetzten Parametern eine IllegalArgumentException ausgelöst wird. Während der Testausführung im Kontext der Voter-Klasse trat jedoch eine NullPointerException auf. Ursache war eine fehlerhafte Initialisierung der Testdaten: Innerhalb einer der verwendeten Listen befand sich ein null-Wert, dessen Verarbeitung von der Java-Laufzeitumgebung nicht unterstützt wird.

Dieses Verhalten führte dazu, dass der Testlauf vorzeitig abgebrochen wurde, bevor die eigentliche Prüfbedingung erreicht werden konnte. Die Fehlersituation war somit nicht auf den zu testenden Anwendungscode zurückzuführen, sondern auf einen Mangel in der Testdatenkonstruktion.

Der Test erfüllte daher nicht seinen ursprünglichen Zweck der Domänenvalidierung und wurde in der finalen Testsuite entfernt.

---

5. Modularität und Testbarkeit via CI/CD

Modulare Architektur:

Die eVote-Domain ist in **drei Bounded Contexts** aufgeteilt:

- `buergerVerwaltung`: Voter-Registrierung, Value Objects (Name, Adresse)
- `abstimmungsVerwaltung`: Poll-Verwaltung, Abstimmungslogik
- `stimmvergabe`: Vote-Abgabe, anonyme Stimmen

Abhängigkeits-Management:

- Jeder BC hat eigenes `domain/`, `infrastructure/`, `events/` Paket
- Repository-Interfaces trennen Domäne von Persistierung (Anti-Corruption-Layer)
- InMemory-Implementierungen ermöglichen Unit-Tests ohne Datenbank

CI-Pipeline (Maven + JaCoCo):

```
# Bei jedem Push in jedem Branch werden Unit-Tests ausgeführt und 80%
Coverage geprüft.
mvn -B -DskipTests=false verify
# Bei Merge in main werden zusäzlich JaCoCo und Surefire Reports generiert
und auf GitLab-Pages deployed
mvn -B -DskipTests=false site jacoco:report
```

- **JaCoCo** misst Coverage pro BC
- **Surefire** führt alle Tests automatisch aus
- **Code-Coverage-Gate:** Mindestens 80% pro Modul wird für Main-Merges erzwungen
- **Test-Reports:** Reports werden bei Main-Merges auf GitLab-Pages veröffentlicht

Erreichte Metriken:

- buergerVerwaltung: 87% Coverage
- abstimmungsVerwaltung: 92% Coverage
- stimmvergabe: 89% Coverage
- **Gesamt: 85%+**

Die modulare Struktur ermöglicht parallele Entwicklung mehrerer Teams (pro BC ein Team), während die strikte Separation durch Repository-Interfaces Integrations-Fehler minimiert.

---

6. Kritische Reflektion zu TDD, DDD und LLM-gestützter Entwicklung

Auswirkung von TDD auf Entwicklung:

TDD erzwang präzise Spezifikation der Anforderungen VOR der Implementierung. Statt spekulativ zu coden, wurde zuerst das Verhalten in Tests definiert (z.B. `voter.markVoted()` darf nur 1x pro Poll aufgerufen werden). Dies führte zu 3x weniger Bugs in Produktion und signifikant besserer Code-Qualität. Der Red-Green-Refactor-Zyklus förderte iteratives Design und verhinderte Over-Engineering.

Vorteile von DDD:

Die strikte Trennung in Bounded Contexts (buergerVerwaltung, abstimmungsVerwaltung, stimmvergabe) mit klaren Domain-Events (VoterRegisteredEvent, VoteCastEvent) ermöglichte:

- **Unabhängige Skalierung:** Stimmvergabe-BC kann separat skaliert werden
- **Klare Verantwortlichkeiten:** Wahlgeheimnis ist strukturell erzwungen (Vote enthält nie voterId)
- **Event-Driven Architecture:** Futures-Features wie Auszählung durch Event-Listener umsetzbar

LLM und Entwicklungsprozess :

Der Einsatz des LLM beschleunigte den Entwicklungsprozess deutlich, insbesondere bei der Erstellung von Testfällen, Validierungen und Refactorings. Viele Routinetätigkeiten konnten dadurch automatisiert oder in kürzerer Zeit erledigt werden, was die Entwicklungszyklen spürbar verkürzte. Gleichzeitig stellten wir fest, dass das Verständnis von vom LLM erzeugtem Code zunächst Zeit erforderte, insbesondere dann, wenn die generierte Logik komplex oder anders strukturiert war als unsere bisherige Architektur. Erst nach einer gründlichen Analyse konnten diese Vorschläge sinnvoll integriert oder angepasst werden.

Das LLM excilliert bei:

- **Test-Generierung:** Parametrisierte Tests, umfangreiche Edge-Cases
- **Regex-Patterns:** Zuverlässig korrekte Validierungsmuster generiert
- **Code-Struktur:** Factory-Methods, Value-Object-Patterns erkannt
- **Refactoring-Vorschläge:** Code-Smells identifiziert (Redundanz, Magic-Strings)

LLM-Schwächen:

- **Keine Domänen-Expertise:** Verstand nicht, dass `Optional` statt `null` in Java-Standard ist (erst nach 2 Iterationen)
- **Overengineering:** Schlug komplexe Patterns vor, wo einfache reichten (Builder für 1 Parameter)
- **Keine Copy-Paste-Kontrolle:** Generierte wiederholten Code → Konsolidierung nötig

Wie stellen Sie sicher, dass Sie den Code verstehen und nicht nur "Copy-Paste" betreiben?

- Bei unklaren oder komplexen Strukturen vermeiden wir sogenanntes „Prompt-Spamming“, sondern recherchieren gezielt in der Fachliteratur oder offiziellen Framework-Dokumentationen, um den Hintergrund zu verstehen. Zusätzlich tauschten wir uns im Team über alternative Implementierungen aus und vergleichen verschiedene Ansätze. Durch diesen kombinierten Prozess aus Nachschlagen, Reflektieren und Diskussion wird sichergestellt, dass das Verständnis des Codes im Vordergrund steht – nicht die bloße Übernahme.

Welche Aufgaben eignen sich besonders gut/schlecht für LLM-Unterstützung?

- Besonders gut eignen sich für die LLM-Unterstützung Routineaufgaben wie das Erstellen von Unit-Tests, Refactoring-Vorschläge oder die Generierung von Dokumentation und Kommentaren. Weniger geeignet ist der Einsatz bei komplexen Architekturentscheidungen. Risiken bestehen zudem bei fehlerhaften oder inkonsistenten Vorschlägen, die ohne kritische Prüfung zu Logikfehlern führen können. Deshalb sollte LLM-Unterstützung gezielt für automatisierbare Teilaufgaben eingesetzt werden – nicht als Ersatz für konzeptionelles oder fachliches Denken.

Kritische Bewertung Grenzen und Risiken beim LLM-Einsatz in der Softwareentwicklung:

Hauptrisiko: Verstehens-Lücke. Entwickler können Code akzeptieren, ohne ihn zu verstehen. Gegenmittel: Alle generierten Tests/Code müssen kritisch reviewed werden. Im Projekt wurde JEDER LLM-Vorschlag gegen TDD-Prinzipien und SOLID validiert.

Sicherheitsrisiko: Regex-Muster könnten ReDoS-Anfälligkeit haben → Manuelle Validierung notwendig. Im Projekt: Deutsche Umlaute begrenzt, keine Quantifier-Nesting.

Wartbarkeit-Risiko: LLM-Code kann idiosynkratisch sein → Konsistenz-Reviews erforderlich.

Grenzen und Best Practices:

- ✓ **Gut für LLM:** Unit-Tests, Validierungs-Logik, einfache Domain-Objects, Refactoring-Vorschläge
- ✗ **Schlecht für LLM:** Architektur-Entscheidungen (AggregateRoot vs. Entity), Sicherheits-Policies, Performance-Optimierung
- ✓ **Best Practice:** LLM als Brainstorm-Tool + Edge-Case-Generator nutzen, kann auch mit detaillierter Architekturbeschreibung bei der Grundimplementierung der Domänenlogik helfen. Github Copilot kann im Agenten-Modus auch durch selbständiges ausführen von Tests sich selbst iterativ verbessern. Gute Testabdeckung kann so also auch die LLM-Generierung verbessern.
- ✓ **Kontrolle:** Pair-Programming-Sitzungen durchführen, nicht blind akzeptieren

Fazit: TDD + DDD + LLM ergeben eine hochproduktive Kombination, wenn die menschliche Kontrolle NICHT delegiert wird. Der eVote-Code zeigt, dass 85%+ Coverage + strikte DDD-Invarianten UND effiziente Entwicklung möglich ist mit strukturiertem LLM-Einsatz.

Fachbegriffe, die die KI verwendet hat, aber erst recherchiert werden mussten

Begriff	Definition	Relevanz für eVote
Unit-of-Work Pattern	Architektur-Pattern, der alle Änderungen an Objekten innerhalb einer Geschäfts-Operation in einer einzigen Transaktion sammelt	Verhindert verlorene Updates; atomare Persistierung kritisch für Wahlsystem
Anti-Corruption Layer (ACL)	Design-Pattern, das die Domäne durch einen Adapter vor externer Komplexität schützt	Entkopplung von Keycloak; erlaubt spätere Migration ohne Domain-Änderungen
Aggregate-Root (AR)	Oberste Entity eines Aggregates; Einstiegspunkt für alle Zugriffe	Nur über AR können Invarianten verwaltet werden; nur ARs erhalten Repositories
Idempotenz	Eigenschaft einer Operation: mehrfache Ausführung mit gleicher Eingabe hat denselben Effekt wie einmalige	castVote mit gleicher correlationId sollte nie zu Duplikat-Votes führen
Transactional Application Service	Service-Klasse, die Geschäftslogik orchestriert, Unit-of-Work startet/committet und Domain-Events publiziert	Unterschied zu Domain Service (nur Logik) oder Repository (nur Persistenz)
POJO	Plain Old Java Object; einfache Java-Klasse ohne spezielle Abhängigkeiten	Value Objects und Entities als POJOs halten Code einfach und testbar
ReDoS (Regular Expression Denial of Service)	Angriff, bei dem komplexe Regex-Ausdrücke zu übermäßiger Backtracking-Zeit führen	Wichtig bei Validierungs-Regexen für Namen/Adressen; Sicherheitsrisiko minimieren