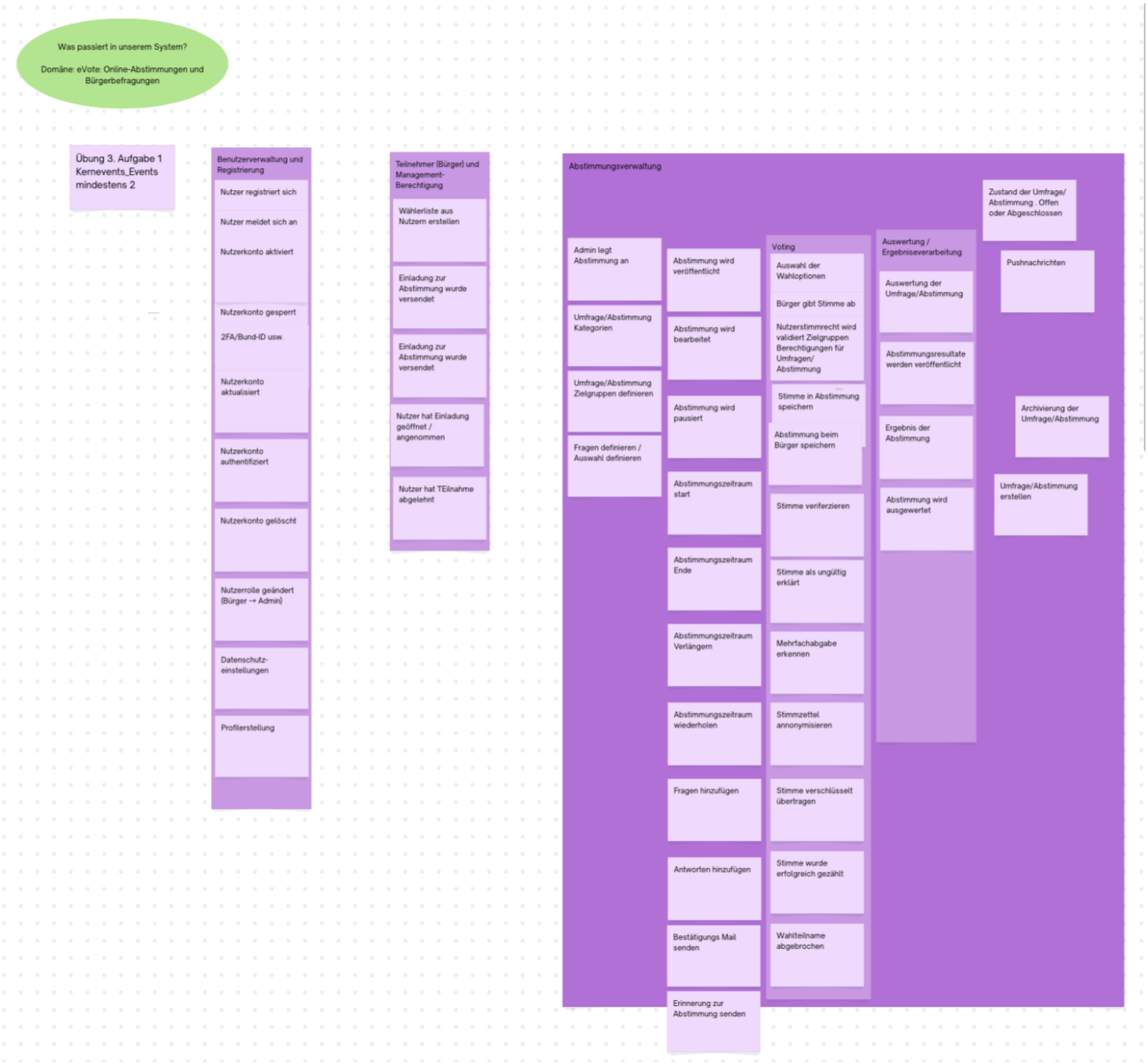
UML & OOD(DDD)

1. Eventstorming

Erste Eventstorming Session mit grober Einteilung / Sortierung in Bounded Contexts:



Die wesentlichen Events:

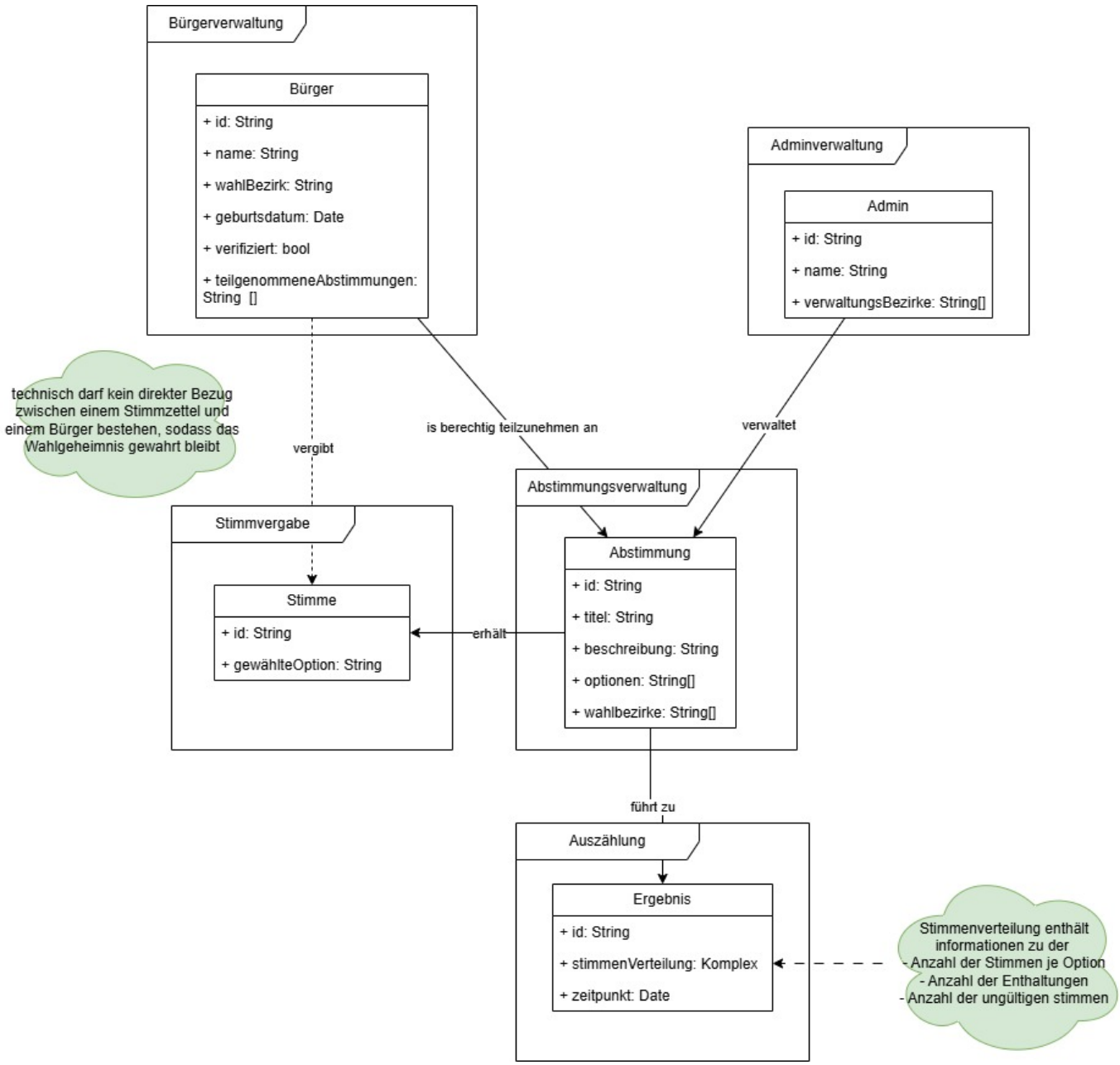
- Bürger registriert: Ein neuer Bürger wird in das System aufgenommen.
- Abstimmung erstellt: Abstimmung wird aufgesetzt mit Fragen und einem Zeitraum.
- Bürger zur Abstimmung freigeschaltet: Bürger wird verifiziert.
- Bürgerstimme abgegeben: Bürger hat Auswahl getroffen.
- Abstimmungsergebnis berechnet: Abstimmungsergebnis wird berechnet zu einem Zeitpunkt X.
- Abstimmungsergebnis publiziert: Ergebnis wird nach Ablauf des Zeitraums veröffentlicht.

2. Domänenmodell / Bounded Context erstellen

Kern-Entitäten:

- Bürger (id: String, name: String, wahlbezirk: String, geburtsdatum: Date, verifiziert: bool, teilgenommeneAbstimmungen: String[])
- Admin (id: String, name: String, verwaltungsbezirke: String[])
- Abstimmung (id: String, titel: String, beschreibung: String, optionen: String[], wahlbezirke: String[], startDatum: Date, endDatum: Date)
- Stimme (id: String, string gewählte option, abstimmung: fk)
- Abstimmungsergebnis (id: String, stimmenverteilung: Stimmenverteilung, erstelltUm: Date, abstimmung: fk)

Klassendiagramm nach DDD:



3. Bounded-Context identifizieren::

Die Domäne des eVotes lässt sich in mehrere Bounded Contexts unterteilen:

- Bürgerverwaltung: Verwaltet Bürgerdaten (Registrierung, Aktualisierung, Löschung ect.) Stellt sicher, dass jeder Bürger nur einmal registriert wird und erst nach erfolgreicher Verifikation freigeschaltet ist.
- Abstimmungsverwaltung: Verwaltung von verschiedenen Abstimmungen. Validiert Start- und Endzeitraum und erlaubt die Stimmenabgabe nur im aktiven Zeitraum, nimmt Stimmen entgegen.
- Stimmvergabe: Erfasst die Stimmen der verifizierten Bürger. Sichert ab, dass jeder Bürger pro Abstimmung nur eine gültige Stimme abgibt.
- Auszählung: Berechnet die Gesamtergebnisse pro Abstimmung. Berechnet und unterscheidet zwischen Gesamt-, gültigen und ungültigen Stimmen sowie Enthaltungen.

4. Entitäten und Aggregates

Bürgerverwaltung: Entität: Bürger Aggregate: Bürger (enthält Daten wie Name, Adresse, Geburtsdatum, teilgenommeneAbstimmungen, Verifizierungsstatus, ID-Nummer)

Abstimmungsverwaltung: Entität: Auszählung Aggregate: Auszählung (Enthält Daten zur Abstimmung wie Art/Titel der Abstimmung, Abstimmungszeitraum, Abstimmungsstatus, ...)

Stimmvergabe: Entität: Stimme Aggregate: Stimme (Enthält alle Daten zur Stimme, wie die ausgewählte Option und die zugehörige Abstimmung)

Auszählung: Entität: Wahlergebnis Aggregate: Wahlergebnis (Enthält die summierte Stimmenanzahl je Option und Anzahl der Enthaltungen zu einem Zeitpunkt X)

5. Domain Services und Repositories

Domain Services:

Bürger-Service: Registriert neue Bürger, verifiziert die Identität und prüft die Wahlberechtigung.

```
registriereBürger(name: String, adresse: String, geburtsdatum: Date):
Bürger
verifiziereBürger(bürgerId: String): void
prüfeWahlberechtigung(bürgerId: String): bool
```

Abstimmung-Service: Erstellt und verwaltet Abstimmungen, kontrolliert den Abstimmungsstatus und schaltet berechtigte Bürger zu Abstimmung frei.

```
erstelleAbstimmung(titel: String, beschreibung: String, optionen: String[],
wahlbezirke: String[], startDatum: Date, endDatum: Date): Abstimmung
schalteBürgerFrei(bürgerId: String, abstimmungId: String): void
prüfeAbstimmungsStatus(abstimmungId: String): AbstimmungsStatus
```

Stimmvergabe-Service: Nimmt Stimmen entgegen, prüft ihre Gültigkeit und speichert sie.

```
checkVoteValidity(bürgerId: String, abstimmungId: String): bool
castVote(bürgerId: String, abstimmungId: String, option: String): void
```

Ergebnis-Service: Zählt und aggregiert Stimmen, berechnet das Gesamtergebnis und veröffentlicht das Ergebnis.

```
zählStimmen(abstimmungId: String): Wahlergebnis
publiziereErgebnis(abstimmungId: String): void
```

Repositories:

Bürger Repository: Verwaltet die Persistenz der Bürgerdaten

```
findeBürgerMitId(String id)
findeAlle()
speicherBürger(Bürger bürger)
```

Abstimmung Repository:

```
findeMitId(String id), findeAktiveAbstimmungen(), speicher(Abstimmung
abstimmung)
```

Stimme Repository:

```
findeMitAbstimmung(String abstimmungId), speichern(Stimme stimme)
```

Ergebnis Repository: Verwaltet die Persistenz der (Ergebnis-) Daten

```
findeErgebnisMitId(String id)
speicherErgebnis(Wahlergebnis wahlergebnis)
```

6. Implementierungsstrategie

1. Test-first-Ansatz (TDD):

Zuerst werden Unit-Tests für die zentralen Geschäftsregeln und Aggregatzuständen geschrieben. Das Ziel ist es sicherzustellen, dass jede Domänenlogik korrekt funktioniert, bevor Implementierungen erfolgen.

Beispiel für Testfälle:

- Bürger-Aggregat: Bürger darf nur mit gültiger Adresse erstellt werden `(adresse != null && adresse.length > 5)` Bürger kann erst abstimmen, wenn er verifiziert/freigeschaltet ist.
- Abstimmung-Aggregat: Eine Abstimmung darf nur erstellt werden, wenn das Startdatum kleiner als das Enddatum ist. Nur berichtigte Stimmen werden gewertet.
- Stimmenvergabe Service: Ein Bürger kann nur eine oder keine Stimme pro Abstimmung abgeben:
    - Ist Stimme gültig
    - Doppelte Stimmen führen zu einer Exception
- Auszählung-Service:
    - Stimmen werden gezählt und als gültig / ungültig klassifiziert

Beispiel (JUnit-Test)

```
@Test
void  throwsExceptionOnInvalidAddress(){
    assertThrows(IllegalArgumentException.class, () ->
    new Bürger ("1", "Max Mustermann, "" )
);
}
```

2. Entitäten und Aggregates umsetzen:

Jede Entität (Bürger, Abstimmung, Stimme, Wahlergebnis) wird als eigene Java-Klasse umgesetzt. Aggregate kapseln ihre interne Businesslogik. **Die Implementierung erfolgt schrittweise anhand der bestehenden Tests.**

3. Domain Services umsetzen

Nach stabilen Aggregaten werden dann die Services umgesetzt. Sie koordinieren vor allem die Logik über mehrere Aggregate hinweg und greifen auf Repositories zu um die Geschäftsregeln durchzusetzen.

4. Repositories umsetzen

Mittels JPA werden die Repositories mit einer Datenbank verbunden, um Aggregatzustände dauerhaft speichern und abrufen zu können.

5. Refactoring / Coding Standards:

Ebenso wird der Code refaktoriert und gemäß DDD- und Clean-Code-Prinzip gestaltet. Mit einem Fokus auf Lesbarkeit, Wartbarkeit und klare Verantwortlichkeiten.