DOMAIN-DRIVEN DESIGN - JAK OKIEŁZNAĆ LOGIKĘ BIZNESOWĄ

•••

Bartosz Schiller

Część 1

Co to jest DDD

- Domain-Driven Design - Projektowanie Sterowane Dziedziną
- DDD zostało zaprezentowane i opisane po raz pierwszy przez Erica Evansa w jego książce z 2003 roku
- DDD nie jest frameworkiem, biblioteką, wzorcem architektonicznym ani projektowym
- Jest to przede wszystkim podejście do modelowania i projektowania, miejscami wzbogacone o zalecenia dotyczące implementacji
- Najważniejsza zasada podczas implementacji - kod odzwierciedlający dziedzinę

Każdy system składa się z 2 części: modelu i implementacji.

Model i implementacja muszą być ze sobą zgodne - każda zmiana modelu wymusza zmianę implementacji i na odwrót.

Trochę o modelu

- Model nie ma ściśle zdefiniowanej struktury, zazwyczaj są to diagramy UML i opisy
- W definiowaniu i modyfikacji modelu bierze udział zespół deweloperów i eksperci dziedzinowi. Zaangażowanie całego zespołu deweloperów jest istotnym elementem DDD
- Model ewoluuje wraz z coraz lepszym rozumieniem dziedziny (tzw. iteracyjna refaktoryzacja modelu)
- Jeśli rozmawiamy o czymś z ekspertem dziedzinowym, to powinno trafić to do modelu dziedziny
- Dziedzina dzieli się na poddziedziny, co w implementacji nazywane jest *kontekstem związanym* (bounded context)

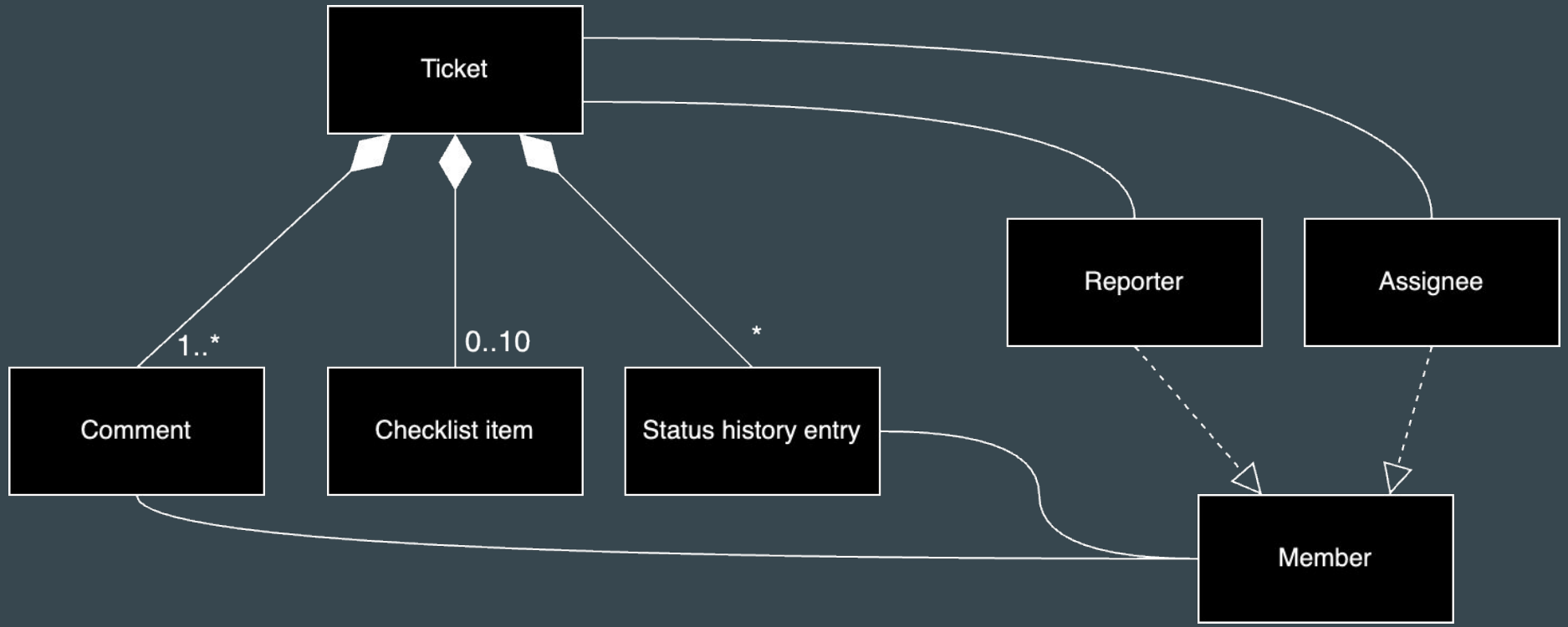
Język wszechobecny (ubiquitous language)

- Komunikacja pomiędzy wszystkimi osobami zaangażowanymi w tworzenie systemu powinna opierać się o zdefiniowaną w **modelu** terminologię
- Nazewnictwo zdefiniowane w **modelu** musi być używane także w **implementacji (w kodzie źródłowym)**
- Język wszechobecny ułatwia komunikację, zapobiega nieporozumieniom, wspomaga nawigację w kodzie źródłowym, jego zrozumienie i powiązanie z modelem

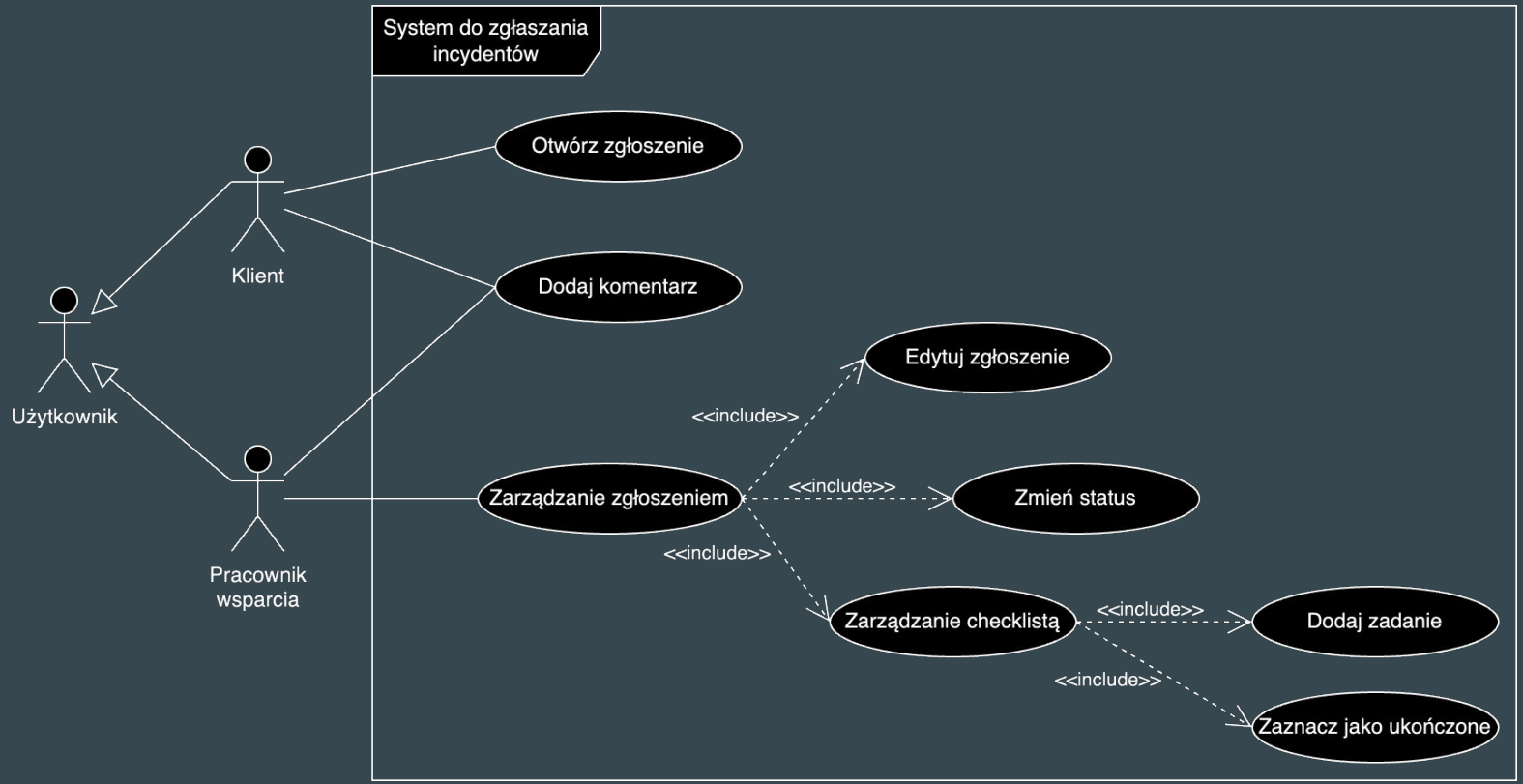
Aplikacja demo - opis klienta

- Prosty system zgłaszania incydentów
- Użytkownicy z dwoma różnymi rolami - klient i pracownik wsparcia
- Klienci zgłaszają incydenty
- Do incydentu automatycznie przypisywany jest pracownik wsparcia
- Incydenty mają identyfikatory w stylu Jiry (<projekt>-<numer>)
- Każdy incydent posiada status opisujący aktualny postęp prac
- Dla każdego incydentu śledzona jest historia zmian statusów
- Incydent zawiera opcjonalną listę zadań - checklistę (maks. 10 zadań). Dziwny limit 10 zadań posłuży nam do zaprezentowania niezmienników (ang. invariants)
- Do incydentu można dodawać komentarze (komunikacja zgłaszającego z pracownikami wsparcia)
- Na potrzeby demo nie implementujemy uwierzytelniania
- Otwarcie nowego zgłoszenia powinno skutkować wysłaniem wiadomości email do przypisanego pracownika wsparcia oraz utworzeniem komentarza

Model aplikacji demo



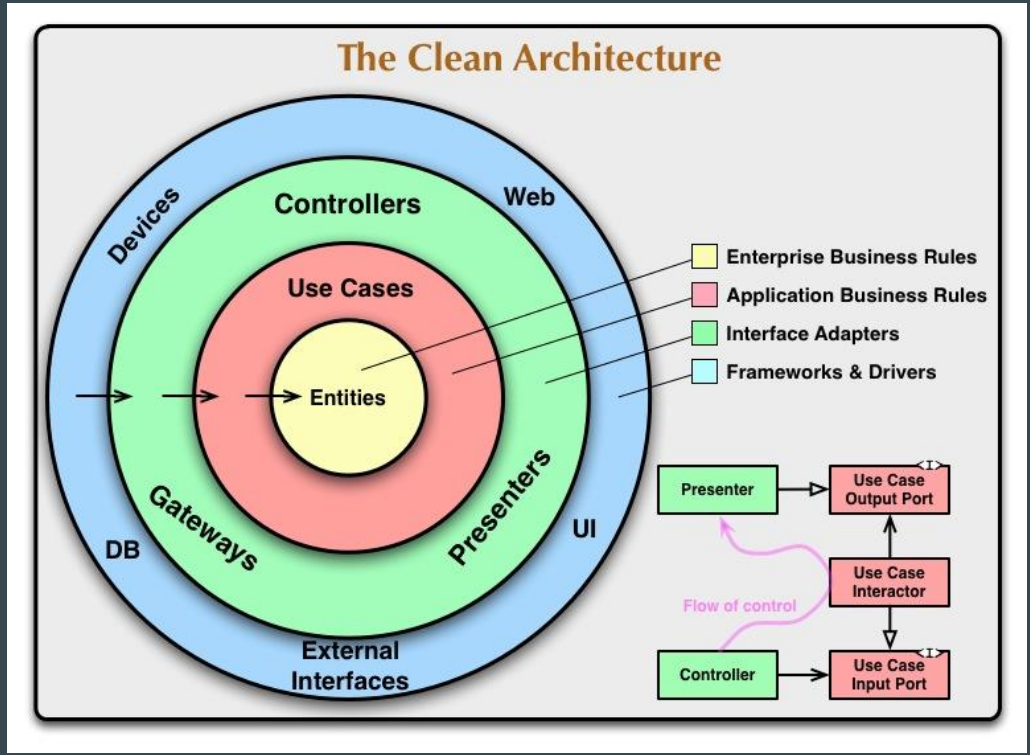
Wybrane przypadki użycia aplikacji demo



IMPLEMENTACJA

Clean Architecture

- Domain (Entities)
- Application (Use Cases)
- Interfaces (Ports)
- Infrastructure
- Warstwy wewnętrzne nie mogą wiedzieć nic o warstwach zewnętrznych zawierających coraz więcej szczegółów implementacyjnych, stąd najważniejsza zasada - typy z danej warstwy mogą być używane tylko w niej lub w warstwie zewnętrznej (The Dependency Rule)



Źródło: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Implementacja dziedziny na przykładzie

- 3 główne składowe - wartości (value object), encje (entity), agregaty (aggregate root)
- Agregaty powinny być jak najmniejsze, a nie trzymać wiele encji
- Nazewnictwo klas (w szczególności dziedzinowych) musi zgadzać się z językiem wszechobecnym, stąd nie nazwiemy klasy `TicketAggregate` tylko `Ticket`
- Niezmienniki (invariants) - reguły opisujące daną encję, np.:
  - blokada edycji zamkniętych zgłoszeń
  - możliwość dodania maksymalnie 10 zadań do checklisty

Wartość (value object - VO)

Obiekty niemutowalne (immutable), których tożsamość zależy od wartości pól

```java
@Getter
@EqualsAndHashCode
public class TicketId {
    private final String id;

    public TicketId(String id) {
        if (id == null || !id.matches(".*-\\d+")) {
            throw new ValidationResultException("ticketId", "must match regex");
        }
        this.id = id;
    }

    public String getProjectName() { return id.split("-")[0]; }

    public int getTicketNumber() { return Integer.parseInt(id.split("-")[1]); }
}
```

```java
@Getter
@EqualsAndHashCode @AllArgsConstructor
public class StatusHistoryEntry {
    // brak ID

    private final TicketStatus previousStatus;
    private final Member editor;
    private final Instant changeDate;
}
```

```java
// przykład spoza demo
@Getter
@EqualsAndHashCode @AllArgsConstructor
public class Address {
    // brak ID

    private final String street;
    private final int number;
    private final String city;

    public Address withNextApartmentNum() {
        return new Address(street, number + 1, city);
    }
}
```

Encja (entity)

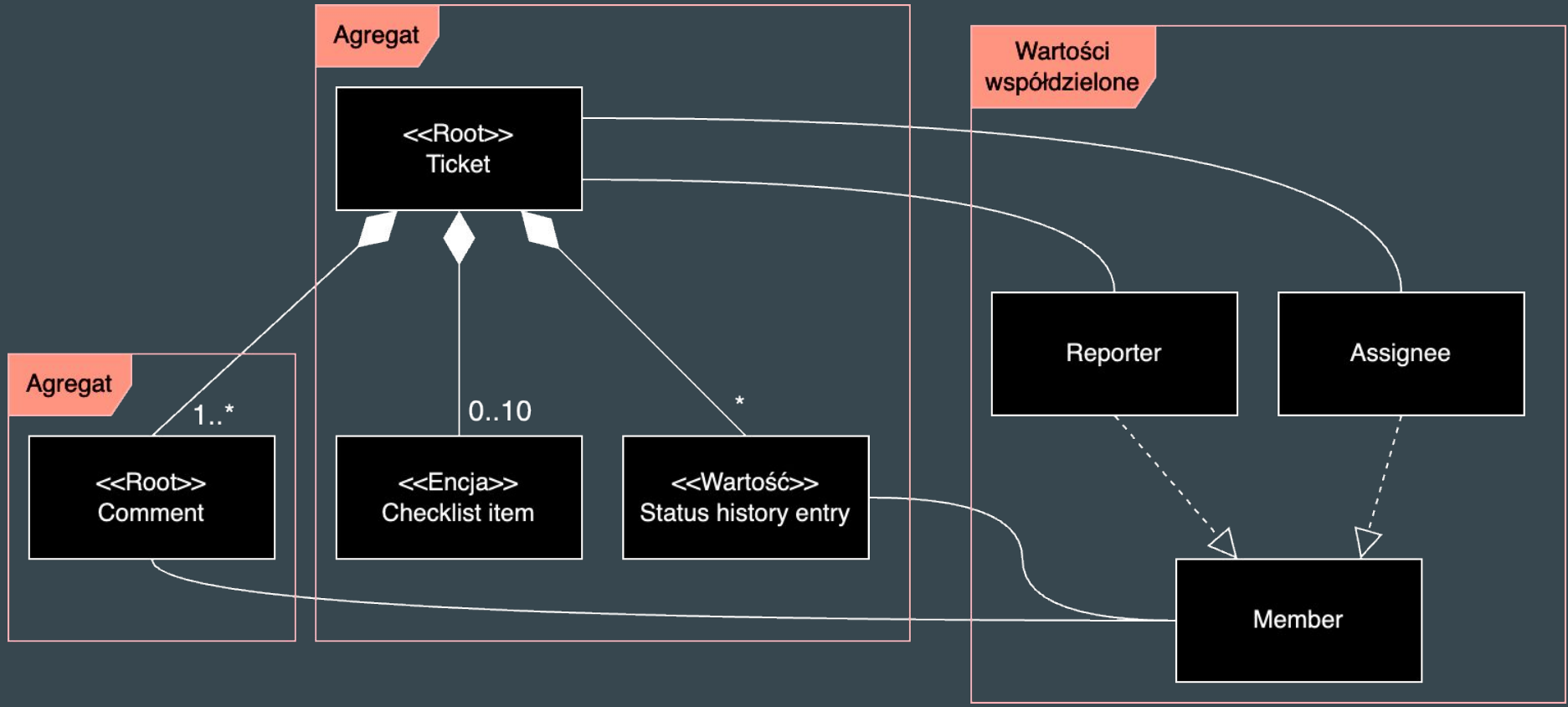
- Encje mogą być dodawane i usuwane tylko poprzez agregat
- Walidacja agregatu
- Walidacja na bazie zewnętrznego walidatora
- Walidacja niezmienników

```java
@Getter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@AllArgsConstructor(access = AccessLevel.PROTECTED)
// wszystkie metody biznesowe są protected, aby mogły być wykonane tylko przez agregat
public class ChecklistItem {
    @EmbeddedId
    @EqualsAndHashCode.Include
    private ChecklistItemId id;
    private TicketId ticketId;
    private String content;
    private boolean checked;

    protected void edit(String content) {
        if (StringUtils.isBlank(content)) {
            throw new ValidationResultException("content", "must not be blank");
        }
        this.content = content.trim();
    }

    protected void check() {
        this.checked = true;
    }

    protected void uncheck() {
        this.checked = false;
    }
```

Agregat (aggregate root)

- Agregat jest encją, która nie należy do innych agregatów i encji (czyli w drzewie encji i wartości, korzeń nazwiemy agregatem)
- Może przechowywać referencje do innych agregatów tylko przez ID
- Po wczytaniu z bazy danych agregat powinien być kompletny, tj. zawierać wszystkie wartości i encje (ew. może wykorzystywać lazy loading z JPA).
- Wszystkie operacje biznesowe na agregacie, zależnych encjach i wartościach muszą odbywać się właśnie w agregacie (ew. w serwisach dziedziny, ale o tym później) w celu wymuszenia niezmienników

Model z podziałem na agregaty



Definiowanie podziału na agregaty

- Podział modelu na agregaty może nie być oczywisty i może zmieniać się w trakcie projektu. Należy uwzględnić niezmienniki, potencjalną liczbę encji i wymagania dotyczące operacji biznesowych na encjach.
- Dlaczego `Comment` jest agregatem, a nie encją w agregacie `Ticket`
  - komentarze nie są nam potrzebne do żadnych operacji biznesowych w zgłoszeniu
  - komentarzy może być bardzo dużo, problematyczne byłoby ich wczytywanie przy każdym ładowaniu agregatu `Ticket`
  - Jak w takim razie wymusić niezmiennik “dodanie komentarza do zamkniętego zgłoszenia nie jest dozwolone”?

```java
@Getter
@AllArgsConstructor(access = AccessLevel.PRIVATE)
public class Ticket extends DomainEntity {
    private TicketId id;
    private String title;
    private String description;
    private Reporter reporter;
    private Assignee assignee;
    private TicketStatus status;
    private List<StatusHistoryEntry> statusHistory;
    private List<ChecklistItem> checklist;
    private Instant createdDate;
    private Instant updatedDate;

    public void updateTitle(String title) {
        if (this.isClosed()) {
            throw new ValidationResultException("ticket", "is closed");
        }

        if (StringUtils.isBlank(title)) { throw new ValidationResultException("title", "is blank"); }

        this.title = title;
        addDomainEvent(new TicketTitleUpdated(id, title));
    }
```

```java
    // instancje agregatów tworzymy przy pomocy metod wytwórczych (factory method) lub fabryk (factory)
    public static Ticket open(String title, String description, Reporter reporter, TicketIdGenerator idGenerator, AssigneeSelectorService assigneeSelector) {
        validateNotBlank(title, "title");
        validateNotBlank(description, "description");
        validateNotNull(reporter, "reporter");

        var assignee = assigneeSelector.select();
        if (assignee == null) {
            throw new IllegalStateException("Cannot assign null assignee");
        }
        Ticket t = new Ticket(
                idGenerator.generateNext(),
                title, description,
                reporter, assignee,
                TicketStatus.RECEIVED,
                new ArrayList<>(), new ArrayList<>(),
                Instant.now(), Instant.now()
        );
        ticket.addDomainEvent(new TicketOpened(t.id, t.title, t.assignee, t.reporter));
        return ticket;
    }
```

```java
    // przykład metody dodającej nową wartość (VO) do listy
    public void changeStatus(TicketStatus newStatus, Member editor) {
        if (this.isClosed()) {
            throw new ValidationResultException("status", "already closed");
        }
        if (!status.canTransit(newStatus)) {
            throw new ValidationResultException("status", "transition not allowed");
        }

        validateNotNull(editor, "editor");

        var prevStatus = this.status;
        this.status = newStatus;
        var historyEntry = new StatusHistoryEntry(prevStatus, editor, Instant.now());
        statusHistory.add(historyEntry);

        addDomainEvent(new StatusHistoryEntryAdded(prevStatus, status, editor,
historyEntry.changeDate()));
    }
```

```java
    // instancję agregatu Comment tworzymy z poziomu agregatu Ticket - możemy zweryfikować
    // niezmienniki i przypisać identyfikator agregatu Ticket do Comment
    public Comment comment(Member author, String content) {
        if (isClosed()) {
            throw new ValidationResultException("ticket", "is closed");
        }
        if (author == null) {
            throw new ValidationResultException("author", "must not be null");
        }
        if (StringUtils.isBlank(content)) {
            throw new ValidationResultException("content", "must not be blank");
        }

        var comment = new Comment(
                new CommentId(Ulid.fast().toLowerCase()),
                getId(),
                author,
                content,
                Instant.now()
        );
        comment.addDomainEvent(new CommentAdded(comment.getId()));
        return comment;
    }
```

```java
    // edycja agregatu Comment z poziomu agregatu Ticket w celu wymuszenia niezmiennika
    public void editComment(Comment comment, String content) {
        if (isClosed()) {
            throw new ValidationResultException("ticket", "is closed");
        }

        comment.edit(content);
    }
```

```java
    // Dodanie encji ChecklistItem do agregatu Ticket. Zgodnie z zasadą, wszystkie operacje
    // na encjach muszą być wykonywane w ramach agregatu.
    public ChecklistItemId addChecklistItem(String content) {
        if (isClosed()) { throw new ValidationResultException("ticket", "is closed"); }
        if (checklist.size() >= 10) {
            throw new ValidationResultException("checklist", "max 10 checklist items allowed");
        }
        if (StringUtils.isBlank(content)) {
            throw new ValidationResultException("content", "is blank");
        }

        var item = new ChecklistItem(
                new ChecklistItemId(Ulid.fast().toLowerCase()),
                id,
                content.trim(),
                false
        );

        checklist.add(item);

        return item.getId();
    }
```

```java
    // wykonanie operacji biznesowej na encji poprzez agregat
    public void checkChecklistItem(ChecklistItemId checklistItemId) {
        if (isClosed()) {
            throw new ValidationResultException("ticket", "is closed");
        }

        ChecklistItem item = findChecklistItem(checklistItemId)
                .orElseThrow(() -> new ValidationResultException("checklistItem", "not found"));
        item.check();
    }

    // dajemy dostęp do listy encji, ale w formie niemutowalnej kolekcji
    public List<ChecklistItem> getChecklist() {
        return Collections.unmodifiableList(checklist);
    }
```

Refaktoryzacja

- Eric Evans przedstawia ciągłą refaktoryzację jako element DDD bazując na metodyce programowania ekstremalnego (XP).
- Iteracyjna refaktoryzacja modelu pociąga za sobą refaktoryzację implementacji
- Tip: refaktoryzację kodu najlepiej przeprowadzać w ramach istniejących zadań. Dodanie osobnych zadań na refaktoryzację zazwyczaj nie działa, gdyż nigdy nie będzie czasu na ich realizację - nie wnoszą nic wartościowego dla project managerów czy ownerów.
- Tip: ważne aby nie marnować zbyt dużo czasu myśląc nad szczegółem implementacyjnym. Jeśli w danym momencie nie wiesz jak coś zaimplementować, zaimplementuj to jakkolwiek i dokonaj refaktoryzacji później.

SERWISY

Serwisy aplikacyjne

- Serwisy aplikacyjne (ang. application services) znajdują się w warstwie use-case'ów
- Odpowiadają za koordynację operacji na dziedzinie w ramach danego use-case'u
  - kontrola transakcji
  - autoryzacja
  - wczytanie agregatu
  - wywołanie metody agregatu lub serwisu dziedziny
  - zapisanie agregatu
  - zapisanie metryk
  - wysłanie logów
  - itp.
- Nie zawierają logiki biznesowej

```java
@Service @Transactional
@RequiredArgsConstructor
public class TicketService {
    private final TicketRepository ticketRepository;
    private final AssigneeSelectorService assigneeSelectorService;
    private final TicketIdGenerator ticketIdGenerator;
    private final MemberRepository memberRepository;
    // ID zwracany z serwisu aplikacyjnego jest w formie stringa, a nie typu dziedzinowego
    public OpenTicketOutput openTicket(String title, String description, CurrentUserInfo userInfo) {
        Reporter reporter = memberRepository.findReporterById(userInfo.id())
                .orElseThrow(() -> new EntityNotFoundException("User not found"));
        var t = Ticket.open(title, description, reporter, ticketIdGenerator, assigneeSelectorService);
        ticketRepository.save(t);
        return new OpenTicketOutput(t.getId().id(), t.getTitle(), t.getAssignee(), t.getReporter());
    }

    // serwis aplikacyjny otrzymuje ID w formie stringa i tworzy ID typu dziedzinowego
    public void editTitle(String ticketId, String title) {
        Ticket ticket = ticketRepository.findById(new TicketId(ticketId))
                .orElseThrow(() -> new EntityNotFoundException("Ticket not found"));

        ticket.updateTitle(title);
        ticketRepository.save(ticket);
    }
}
```

Serwisy dziedzinowe

- Serwisy dziedzinowe (ang. domain services) znajdują się w warstwie dziedziny
- Zawierają logikę biznesową
- Segment dla purystów - jak zapobiegać zanieczyszczeniu warstwy dziedziny adnotacjami Springa:
  - Definiowanie beanów przy pomocy `@Bean` w klasach `@Configuration` w warstwie infrastruktury
  - Definiowanie beanów przy pomocy implementacji `ApplicationContextInitializer<GenericApplicationContext>`
  - Używanie w warstwie dziedziny tylko interfejsów, umieszczenie implementacji w warstwie infrastruktury

```java
// implementacja serwisu w warstwie dziedziny
@Service
@RequiredArgsConstructor
public class TicketIdGenerator {
    private final TicketRepository ticketRepository;

    public TicketId generateNext() {
         int previousTicketNumber = ticketRepository.findLastTicketId()
                .map(TicketId::getTicketNumber)
                .orElse(0);
        return new TicketId("TT-" + (previousTicketNumber + 1));
    }
}
```

```java
// interfejs serwisu w warstwie dziedziny, implementacja w warstwie infrastruktury
public interface AssigneeSelectorService {
   Assignee select();
}
```

BAZA DANYCH

Podejście do implementacji

- Baza danych to tylko szczegół implementacyjny, wszystkie operacje biznesowe odbywają się w warstwie dziedziny
- Podejście dla purystów - SQL
  - Dziedzina wolna od zależności zewnętrznych bibliotek
  - Brak ograniczeń w modelowaniu dziedziny, narzuconych przez np. JPA
  - Więcej powtarzalnego kodu do napisania i utrzymywania (mapowanie encje <-> model dziedziny, zarządzanie kolekcjami)
- Podejście JPA
  - Dziedzina zanieczyszczona zależnościami bibliotek
  - Ograniczenie w modelowaniu dziedziny narzucone przez dostępne adnotacje i sposób ich działania (mogą ujawnić się w późnej fazie projektu)
  - Mniej kodu i praktycznie bezobsługowy zapis i odczyt do/z bazy

JPA a Spring Data JDBC w DDD

- Zalety Spring Data JDBC (nie mylić ze Spring JDBC)
  - prosty ORM nie będący implementacją JPA
  - zaprojektowane z myślą o zastosowaniu z DDD
  - wspiera repozytoria Spring Data i Query Methods
  - integruje się z MyBatis w celu wykorzystywania zapytań SQL
  - zawiera klasę Query i Criteria dające możliwość dynamicznego generowania zapytań
- Dlaczego jednak wolę JPA
  - Możliwość walidacji struktury bazy danych podczas uruchamiania aplikacji (`ddl-auto: validate`)
  - Możliwość wygenerowania metadanych encji przy użyciu `hibernate-jpamodelgen` (lub Querydsl)
  - Wsparcie dla Specyfikacji przez Spring Data JPA
  - Bezobsługowe audytowanie przy pomocy Envers

Dodanie wsparcia dla JPA do agregatów, encji i wartości

- Dodanie konstruktora wymaganego przez JPA `@NoArgsConstructor(access = AccessLevel.PROTECTED)` do wszystkich agregatów, encji i wartości
- Dodanie adnotacji `@EmbeddedId` do pól z identyfikatorem danej encji bazodanowej
- Dodanie adnotacji `@Embeddable` do wszystkich klas wartości
- Dodanie adnotacji `@JoinColumn(nullable = false, ...)` i `@OneToMany(...)` do kolekcji zawierających encje (dzięki `nullable = false` Hibernate ustawi klucz obcy podczas operacji insert, w przeciwnym wypadku podczas insertu ustawi NULL a później wywoła update z kluczem obcym)
- Dodanie adnotacji `@ElementCollection` i `@CollectionTable(...)` do kolekcji zawierających wartości
- Usunięcie modyfikatora `final` z pól w wartościach (VO)

JPA, wartości i @Embeddable

```java
@Getter @EqualsAndHashCode @NoArgsConstructor(access = AccessLevel.PROTECTED)
@Embeddable
public class CustomerAddress {
    @Column(name = "customer_street")
    private String street;
    // lub
    private String customerStreet;

}
```

```java
@Getter @EqualsAndHashCode @NoArgsConstructor(access = AccessLevel.PROTECTED)
@Entity
public class Customer {
    @AttributeOverride(name = "street", column = @Column(name = "customer_street"))
    private CustomerAddress address;
}
```

```
# lub globalna zmiana domyślnych ustawień Hibernate
spring.jpa.hibernate.naming.implicit-strategy=org.hibernate.boot.model.naming.ImplicitNamingStrategyComponentPathImpl
```

JPA, wartości i JSON

```java
@Getter @EqualsAndHashCode @NoArgsConstructor(access = AccessLevel.PROTECTED)
@Entity
public class Customer {
    // Hibernate 6
    @JdbcTypeCode(SqlTypes.JSON)
    // Hibernate 5
    @Type(...)
    // lub
    @Convert(converter = CustomJsonAttributeConverter.class)
    private CustomerAddress address;
}
```

```java
@Bean
public HibernatePropertiesCustomizer jsonFormatMapperCustomizer(ObjectMapper objectMapper) {
   return properties -> properties
           .put(AvailableSettings.JSON_FORMAT_MAPPER, new JacksonJsonFormatMapper(objectMapper));
}
```

Repozytoria

- W clean architecture repozytorium składa się z dwóch części - interfejsu w warstwie dziedziny i implementacji w warstwie infrastruktury
- Repozytorium służy do pobrania całego agregatu z bazy danych
- Zgodnie z zasadami DDD, encje nie mogą mieć repozytoriów, gdyż są zawsze wczytywane w ramach agregatu

```
1 public interface TicketRepository {
2     Optional<Ticket> findById(TicketId id);
3     Page<Ticket> findAll(Pageable pageable);
4     Page<Ticket> findAll(DomainSpecification<Ticket> spec, Pageable pageable);
5     void save(Ticket ticket);
6     Optional<TicketId> findLastTicketId();
7 }
```

Dirty checking w JPA

- W przypadku poprawnie wykorzystywanych transakcji (a w tym pomaga nam DDD), po dokonaniu zmian na encji nie musimy wołać save() aby zapisać zmian, są one wykrywane dzięki mechanizmowi dirty checking - JPA porównuje naszą encję z kopią encji i sprawdza czy jakieś pola zostały zmienione
- Można wywołać save() aby kod był prostszy w zrozumieniu dla mniej doświadczonych, albo chcemy jasno zaznaczyć nasze intencje, ale save() nie ma w tym wypadku żadnego efektu
- Najlepiej ustalić w zespole jaką przyjmujemy konwencję i się jej trzymać
- W przypadku polegania na dirty checkingu uzyskujemy niejako abstrakcję działania na kolekcjach - po zmodyfikowaniu pola w obiekcie, którego referencja przechowywana jest w kolekcji, zmiana jest zachowywana naturalnie automatycznie, nie musimy wołać dodatkowych metod save()
- Osobiście preferuję zawsze wywoływać save(), szczególnie ze względu na zdarzenia dziedziny (o tym później)

Wczytywanie agregatu w podejściu purystycznym

W przypadku korzystania z osobnych encji bazodanowych i klas modelu dziedziny należy użyć dodatkowej metody wytwórczej.

```
1  public class Ticket {
2      private TicketId id;
3      private Member editor;
4      private String label;
5
6      // metoda wołana przez repozytorium
7      public static Ticket fromDatabase(TicketId id, Member editor, String label) {
8          // dodatkowa weryfikacja danych wejściowych
9          return new Ticket(id, editor, label);
10     }
11
12     // wciągamy do dziedziny zależność konkretnej technologii, niekoniecznie zalecane, ale jak kto lubi
13     public static Ticket fromDatabase(ResultSet rs) {
14         return new Ticket(...);
15     }
16 }
```

WZORZEC SPECYFIKACJA

Wzorzec specyfikacja

- wydzielenie reguł biznesowych do reużywalnych klas
- rozszerzony wzorzec może być wykorzystywany zarówno do weryfikacji pojedynczej instancji agregatu jak i odpytywania repozytorium o listę agregatów spełniających regułę

```
1  public interface DomainSpecification<T> {
2      boolean isSatisfiedBy(T entity);
3
4      Specification<T> toQuery();
5
6      default DomainSpecification<T> and(DomainSpecification<T> other) {
7          return new AndSpecification<>(this, other);
8      }
9
10     // operacje or, not itp.
11 }
```

```java
public class IsTicketOverdue implements DomainSpecification<Ticket> {
    private static final Duration overdueAfter = Duration.of(7, ChronoUnit.DAYS);

    @Override
    public boolean isSatisfiedBy(Ticket ticket) {
        if (ticket.getCreatedDate().isBefore(calculateOverdueBefore())
                && !ticket.getStatus().isClosed()) {
            return true;
        }
        return false;
    }

    @Override
    public Specification<Ticket> toQuery() {
        return (root, query, cb) ->
                cb.and(
                        cb.lessThan(
                                root.get(Ticket_.createdDate),
                                calculateOverdueBefore()
                        ),
                        cb.notEqual(root.get(Ticket_.status), TicketStatus.CLOSED)
                );
    }

    private Instant calculateOverdueBefore() { return ZonedDateTime.now().minus(overdueAfter).toInstant(); }
}
```

ZDARZENIA DZIEDZINY

Zdarzenia dziedziny

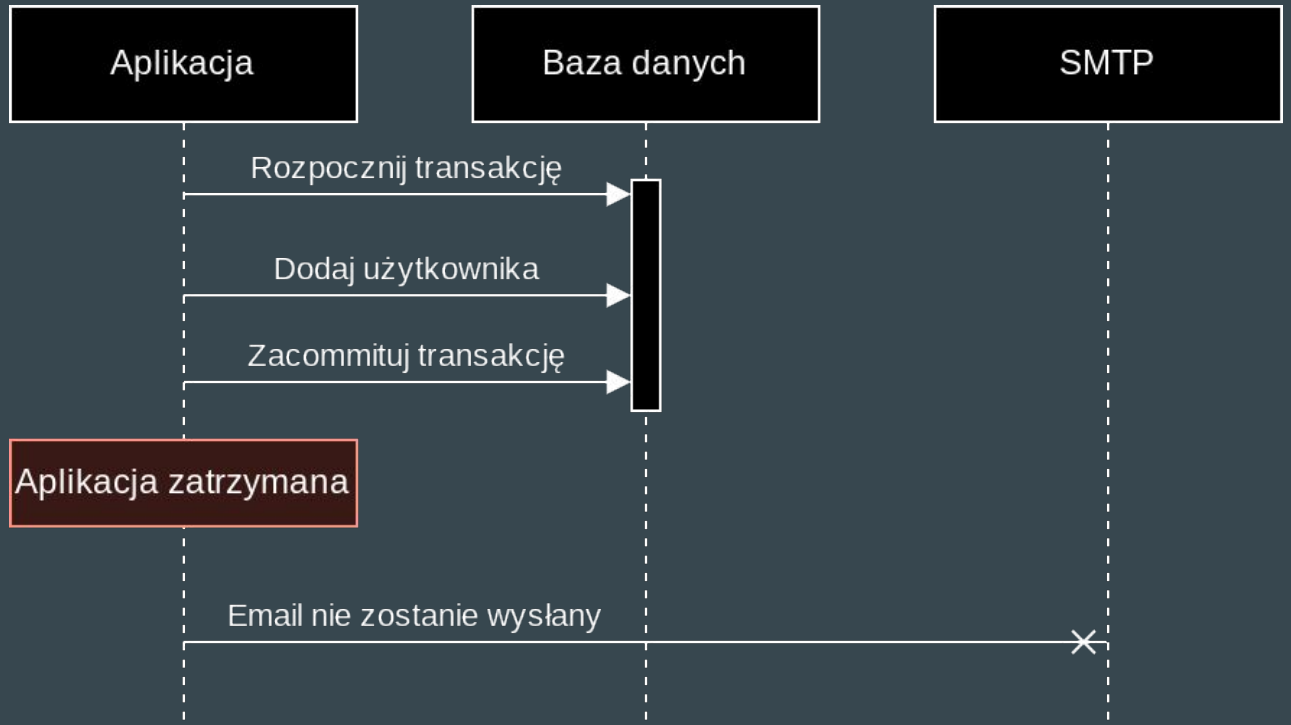
- Eric Evans w swojej książce nie wspomniał o zdarzeniach dziedziny, ale dość szybko stały się one de-facto standardem za sprawą innych developerów
- Pozwalają na odseparowanie zmian na agregatach od efektów ubocznych
  - wykonanie zależnej zmiany na innym agregacie
  - wysłanie maila/notyfikacji push
  - publikacja wiadomości na szynie
  - wywołanie API RESTowego
- Obsługa zdarzeń
  - w ramach transakcji
  - asynchronicznie po zacommitowaniu transakcji (tzw. notyfikacje o zdarzeniach dziedzinowych)
- Zdarzenia informują o czymś co już się wydarzyło, np. `TicketOpened`, `CommentAdded`
- Zdarzenia muszą zawierać informacje potrzebne do ich obsłużenia, ale nigdy nie powinny zawierać instancji agregatu/encji, aby zapobiec dokonywaniu na nich zmian. Mogą za to zawierać wartości (value object), gdyż są one niemutowalne.

Obsługa zdarzeń

- Komunikacja np. z innym serwerem przez REST nie powinna odbywać się w ramach tej samej transakcji bazodanowej, w której modyfikujemy agregat
  - blokujemy zasoby (wątek http, połączenie z bazą) nawet na kilkadziesiąt sekund
  - komunikacja REST i tak nie jest transakcyjna
- API RESTowe możemy zawołać po zacommitowaniu transakcji. Ale co jeśli np. serwer ulegnie awarii po transakcji, ale przed zawołaniem API?
- Outbox pattern wprowadza więcej złożoności i wymaga czasu na implementację, ale gwarantuje obsługę zdarzeń asynchronicznych w modelu at-least-once

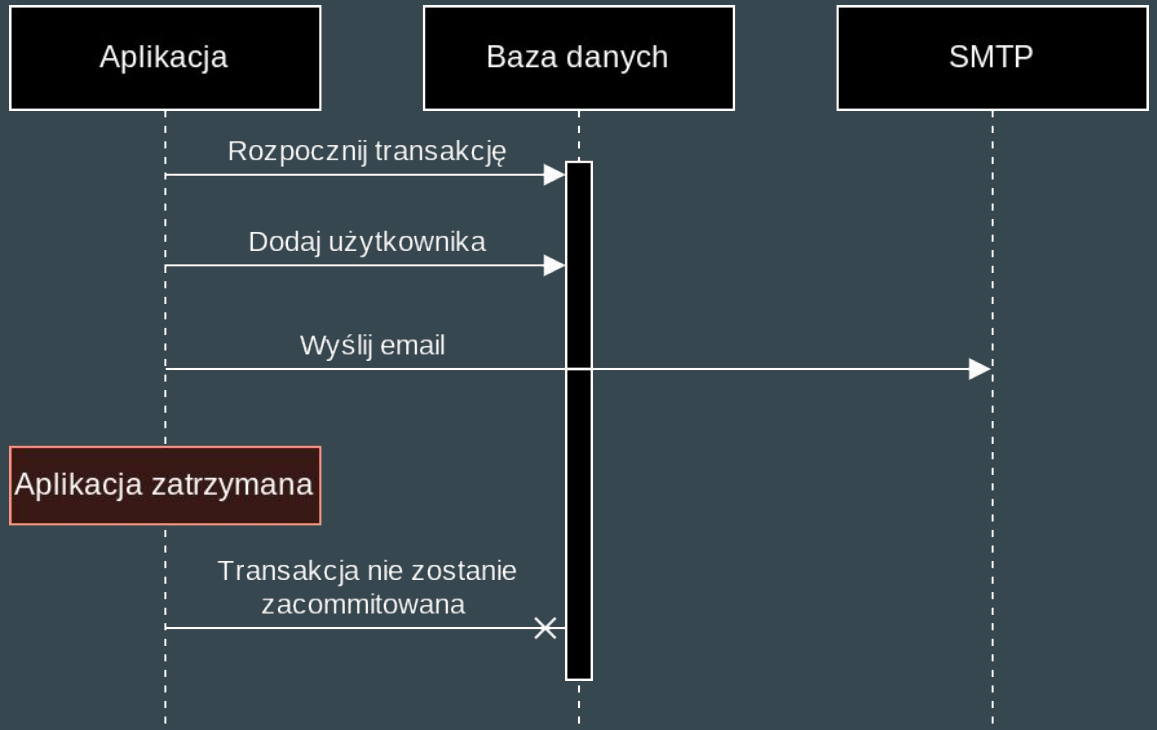
Obsługa zdarzeń - podejście 1.

Obsługa zdarzenia po zacommitowaniu transakcji

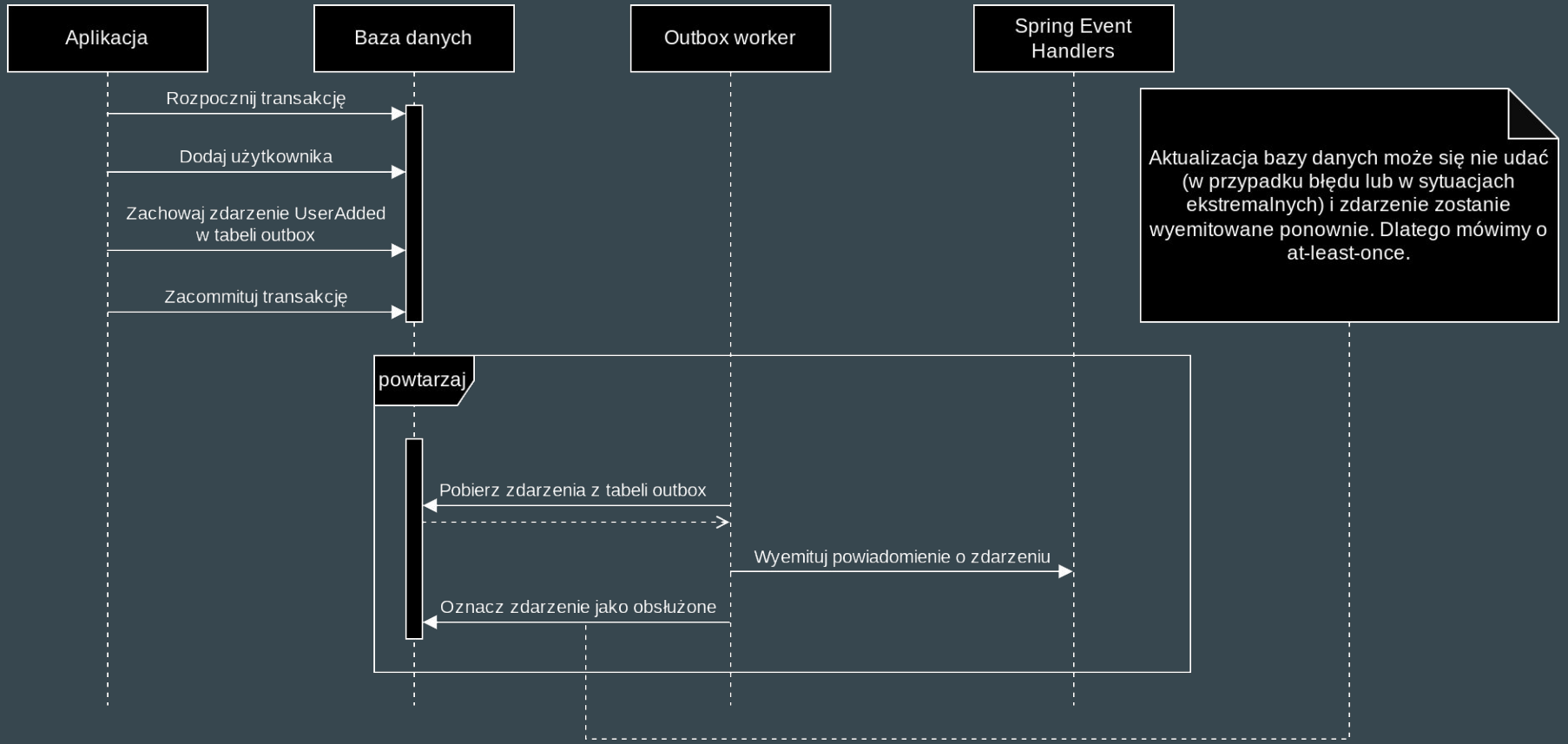


Obsługa zdarzeń - podejście 2.

Obsługa zdarzenia w ramach transakcji



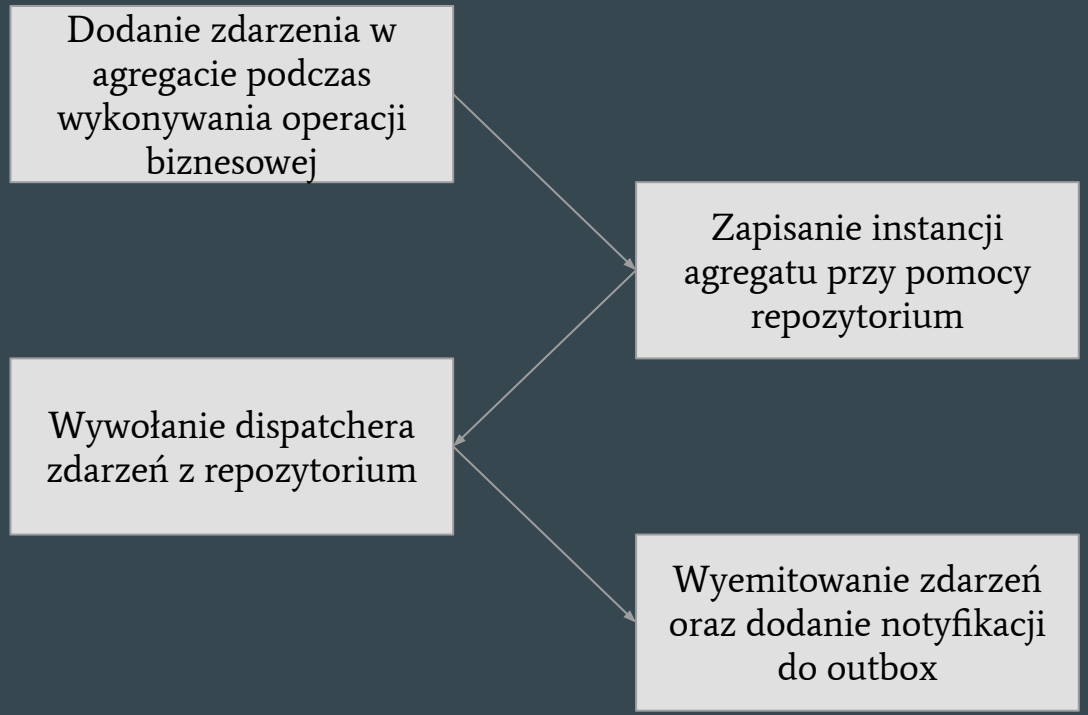
Asynchroniczna obsługa zdarzeń - wzorzec outbox



Wzorzec outbox

- Komunikacja np. z innym serwerem przez REST nie powinna odbywać się w ramach tej samej transakcji bazodanowej, w której modyfikujemy agregat
  - blokujemy zasoby (wątek http, połączenie z bazą) nawet na kilkadziesiąt sekund
  - komunikacja REST i tak nie jest transakcyjna
- Aby wywołać np. API RESTowe, publikujemy taki event po zacommitowaniu transakcji. Ale co jeśli np. serwer ulegnie awarii po transakcji, ale przed wysłaniem eventu?
- Outbox pattern wprowadza więcej złożoności i wymaga czasu na implementację, ale gwarantuje obsługę zdarzeń asynchronicznych w modelu at-least-once

Implementacja zdarzeń dziedziny



Nasza implementacja jest jedną z wielu, można także wykorzystać aspekty, albo zachować listę zdarzeń dziedzinowych w `ThreadLocal`u i wyemitować je w listenerach JPA/Hibernate.

W następnej prezentacji pokażę jakie wsparcie zapewnia Spring Data.

```java
// wszystkie agregaty rozszerzają bazową klasę do obsługi zdarzeń
public abstract class DomainEntity {
    private transient final List<DomainEvent> domainEvents = new ArrayList<>();

    public void addDomainEvent(DomainEvent event) {
        domainEvents.add(event);
    }

    public void clearDomainEvents() {
        domainEvents.clear();
    }

    public List<DomainEvent> getDomainEvents() {
        return Collections.unmodifiableList(domainEvents);
    }
}
```

```java
public record TicketOpened(TicketId ticketId, String title, Assignee assignee,
                           Reporter reporter) implements DomainEvent {
}
```

```java
public class Ticket extends DomainEntity {
    public static Ticket open(String title, String description, Reporter reporter,
            TicketIdGenerator idGenerator, AssigneeSelectorService assigneeSelector) {
        // ...
        Ticket t = new Ticket(...);
        // dodanie nowego zdarzenia do listy zdarzeń
        ticket.addDomainEvent(new TicketOpened(t.id, t.title, t.assignee, t.reporter));
        return ticket;
    }
}
```

```java
// serwis aplikacyjny
public OpenTicketOutput openTicket(String title, String desc, CurrentUserInfo userInfo) {
    // ...
    var ticket = Ticket.open(title, desc, reporter, ticketIdGenerator, assigneeSelectorService);
    // wywołanie metody save będzie skutkować wyemitowaniem zdarzeń dziedziny
    ticketRepository.save(ticket);
    return new OpenTicketOutput(...);
}
```

```java
@Repository
@Transactional
@RequiredArgsConstructor
public class JpaTicketRepository implements TicketRepository {
    // JpaRepository ze Spring Data
    private final TicketQueries ticketQueries;
    private final DomainEventDispatcher domainEventDispatcher;

    @Override
    public void save(Ticket ticket) {
        ticketQueries.save(ticket);
        domainEventDispatcher.publishEntity(ticket);
    }

    // ...
}
```

```java
@Component
@RequiredArgsConstructor
public class DomainEventDispatcher {
    // Springowy bean pozwalający emitować zdarzenia. Domyślnie zdarzenia emitowane są
    // synchronicznie i nie chcemy tego zmieniać.
    private final ApplicationEventPublisher applicationEventPublisher;
    // nasza implementacja wzorca outbox
    private final OutboxService outboxService;

    public void publishEntity(DomainEntity entity) {
        // emitujemy zdarzenia dziedziny w celu obsługi w tej samej transakcji
        entity.getDomainEvents().forEach(applicationEventPublisher::publishEvent);
        // dodajemy zdarzenia do kolejki w celu obsłużenia ich asynchronicznie i w osobnych
        // transakcjach (albo i bez)
        outboxService.addAll(entity.getDomainEvents());
        entity.clearDomainEvents();
    }
}
```

```java
// obsługa zdarzenia w tej samej transakcji
@Component
@RequiredArgsConstructor
// DomainEventHandler to marker interface ułatwiający implementację handlerów
public class AddInitialCommentEventHandler implements DomainEventHandler<TicketOpened> {
    private final CommentService commentService;

    @Override
    @EventListener // Springowa adnotacja
    public void handle(TicketOpened event) {
        commentService.createInitialComment(event.ticketId(), event.reporter());
    }
}
```

```java
@Component
@RequiredArgsConstructor
public class NotifyNewTicketHandler implements DomainEventNotificationHandler<TicketOpened> {
    private final JavaMailSender javaMailSender;
    private final TemplateRenderer renderer;

    @Override
    @EventListener
    // DomainEventNotification to generyczny wrapper dla notyfikacji o zdarzeniach dziedziny
    public void handle(DomainEventNotification<TicketOpened> notification) {
        TicketOpened event = notification.source();
        try {
            var message = javaMailSender.createMimeMessage();
            var messageHelper = new MimeMessageHelper(message);
            var htmlBody = renderer.render("ticket-opened", Map.of("title", event.title(), ...));
            messageHelper.setSubject("Ticket opened " + event.ticketId().id());
            messageHelper.setText(htmlBody, true);
            javaMailSender.send(message);
        } catch (Exception e) {
            log.error("Cannot send email", e);
        }
    }
}
```

```java
// aby Spring poradził sobie z odnalezieniem handlera dla generycznych zdarzeń, używamy
// specjalnego interfejsu ResolvableTypeProvider
public record DomainEventNotification<T>(T source) implements ResolvableTypeProvider {
    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(
                getClass(),
                ResolvableType.forInstance(this.source)
        );
    }
}
```

A na co komu zdarzenia dziedziny?

- Emitując konkretne zdarzenie mamy pewność, że zostanie ono obsłużone zawsze w ten sam sposób przez odpowiedni handler
- Wyzwalanie efektów ubocznych w serwisie aplikacyjnym byłoby wyciekiem logiki biznesowej do warstwy aplikacji, wymagałoby ponadto stałej kontroli czy efekty uboczne są wyzwalane we wszystkich wymaganych miejscach

WYŚWIETLANIE DANYCH

Wyświetlanie danych

- Pobieranie agregatów z bazy i konwertowanie do DTO w warstwie aplikacji
- Zwracanie odpowiedniego DTO z repozytorium (use case optimal query)
- Temat rozwiniemy w następnej prezentacji

ZMIERZAJĄC KU KOŃCOWI

Słowo o uwierzytelnianiu i autoryzacji

- Autoryzacja jest częścią warstwy aplikacji. Najwygodniej zrealizować ją za pomocą adnotacji Spring Security (`@Secured`, `@PreAuthorize`) na metodach w serwisach aplikacyjnych
- Uwierzytelnianie jest zazwyczaj realizowane w warstwie infrastruktury i nie jest częścią dziedziny

Ściągawka

Co znajduje się w jakiej warstwie

- Domain - agregaty, encje, wartości, definicje repozytoriów, serwisy dziedzinowe, specyfikacje
- Application (use-cases) - serwisy aplikacji, obsługa zdarzeń, outbox
- Interfaces - kontrolery, klienty API
- infrastructure - implementacja repozytoriów, konfiguracja Springa i innych bibliotek, generalnie szczegóły implementacyjne

Programowanie to sztuka kompromisu

~Bartolomeo Schillerho

Warto przeczytać

- Eric Evans: Domain-Driven Design (PL i EN) - książka papierowa, ebook
- Vaughn Vernon: Effective Aggregate Design (EN) https://kalele.io/effective-aggregate-design/ - darmowe PDFy
- Kamil Grzybek: Blog (https://www.kamilgrzybek.com/blog) i GitHub (https://github.com/kgrzybek)
- Udi Dahan: Blog (https://udidahan.com/)

W kolejnym odcinku

- CQRS
- Konteksty związane
- Spring Modulith i zdarzenia dziedzinowe
- jMolecules
- I inne

Dziękuję za uwagę

...

Feedback, pytania, groźby:
Slack: @barteksch
Mail: bartosz.schiller@apilia.pl
bartosz.schiller@gmail.com