Architektura sterowana zdarzeniami

Projektowanie i programowanie systemów internetowych II

mgr inż. Krzysztof Rewak

29 listopada 2018

Wydział Nauk Technicznych i Ekonomicznych
Państwowa Wyższa Szkoła Zawodowa im. Witelona w Legnicy

Plan prezentacji

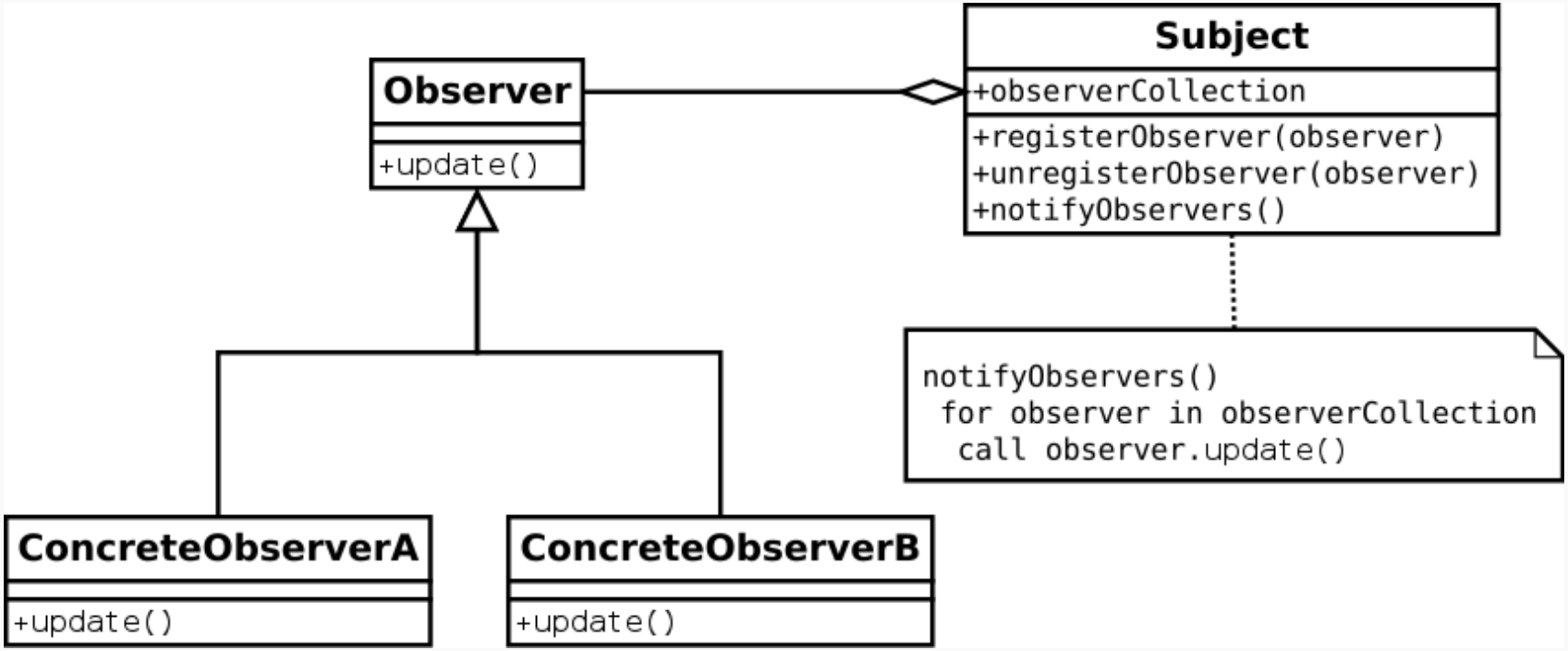
1. Obserwatorzy
2. Zdarzenia
3. Kolejkowanie zdarzeń
4. EDA
5. Podsumowanie

Obserwatorzy

Jesteś obserwowany

Zanim przejdziemy do omawiania aplikacji sterowanych zdarzeniami, chciałem krótko przypomnieć o wzorcu projektowym **obserwator** (ang. *observer*).

UML



Rysunek 1: Diagram UML obserwatora

Wzorzec Obserwator

```
public class User implements Observer {

    @Override
    public void create(Observable observable) {
        observable.createEmptyProfile();
    }

}
```

Obserwatorzy

Współczesne frameworki zazwyczaj umożliwają obserwowanie poprzez podpięcie się do kilku *zdarzeń* towarzyszących modelom.

Obserwatorzy

Przykładowo mogą to być:

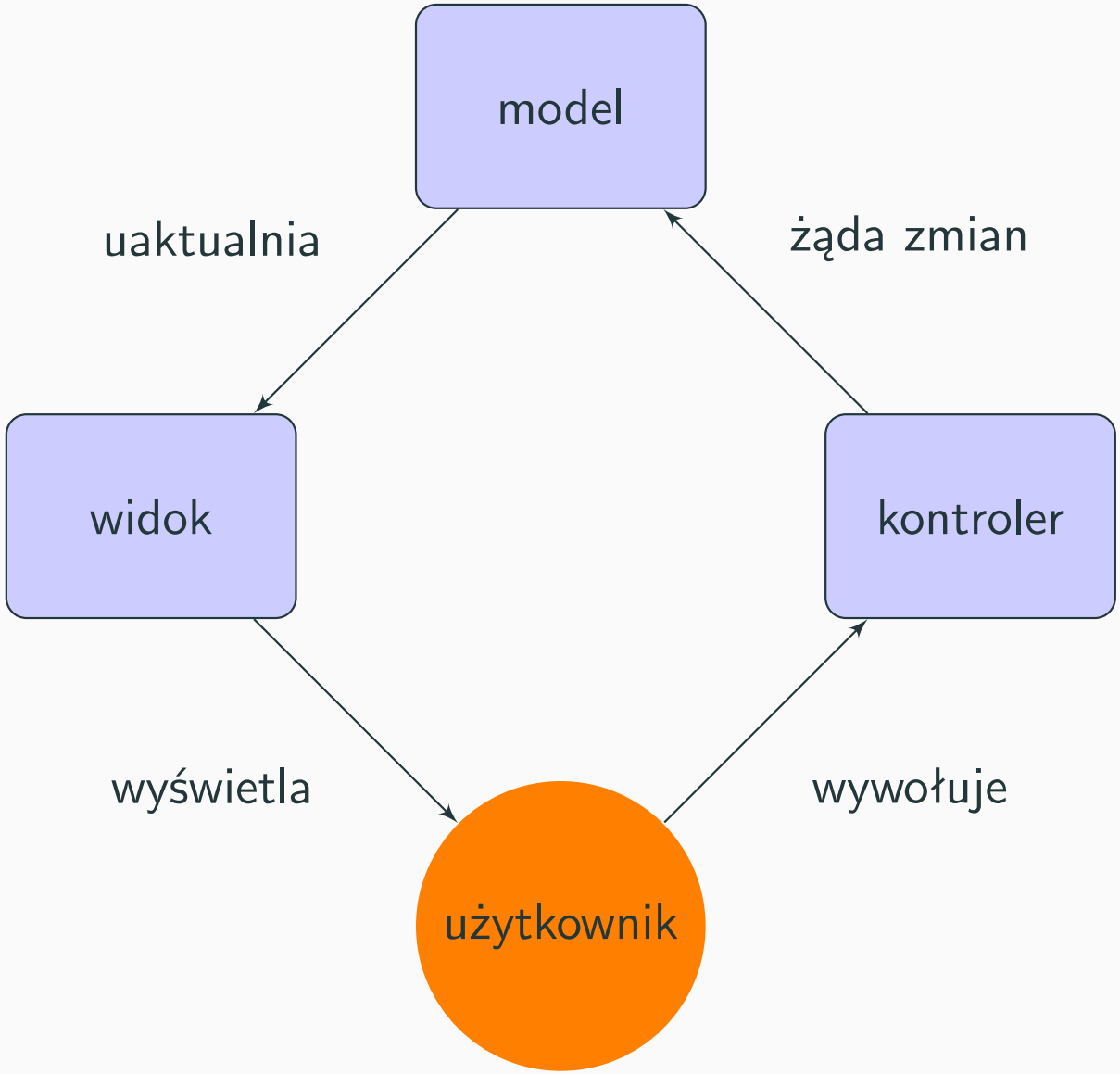
`retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`.

Obserwatorzy

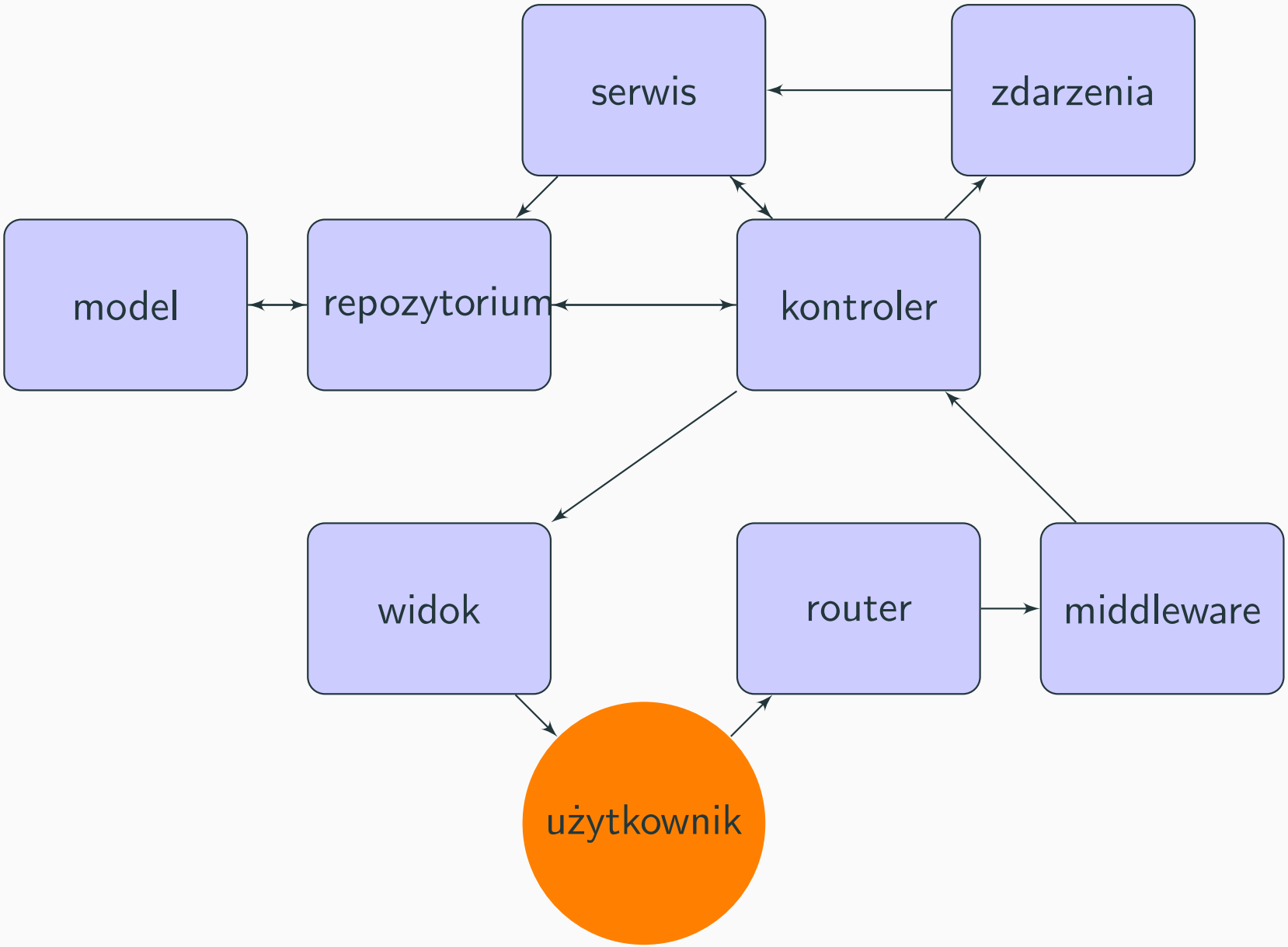
Korzystanie z obserwatorów to pierwszy krok do korzystania z architektury sterowanej zdarzeniami.

Zdarzenia

MVC


model
uaktualnia
żąda zmian
widok
kontroler
wyświetla
wywołuje
użytkownik

MVC... MRMSE?


serwis
zdarzenia
model
repozytorium
kontroler
widok
router
middleware
użytkownik

Zdarzenia?

Proste aplikacje webowe często zachowują się algorytmicznie lub strukturalnie: pobierz zapytanie, zwaliduj dane, uruchom serwis, zapisz coś w bazie danych, wygeneruj powiadomienie, zwróć wynik.

Wiele rzeczy robimy w kontrolerach lub wydzielonych serwisach właśnie w taki sposób.

Co tu nie gra?

```
def signup(request):
  form = SignupForm(request.POST)
  user = form.save(commit=False)
  user.save()

  mail_subject = 'Activate your blog account.'
  message = render_to_string('activate_user.html', {
    'user': user,
    'token': account_activation_token.make_token(user),
  })

  to_email = form.cleaned_data.get('email')
  email = EmailMessage(mail_subject, message, to=[to_email])
  email.send()

  return HttpResponse('Confirm your email address.')
```

Co tam nie gra?

Przede wszystkim rzuca się oczy możliwość stworzenia serwisu rejestrującego użytkownika i wysyłającego email...

... albo dwóch osobnych serwisów to robiących i wywołanych w kontrolerze...

... albo wywołaniu jednego w drugim?

Zdarzenia!

A gdyby zamiast tworzyć (lub przekazywać przez wstrzyknięcie zależności) nową instację serwisu można byłoby krzyknąć do aplikacji: *ej, weź wyślij mejla*?

Lepiej?

```
def signup(request):
  form = SignupForm(request.POST)
  user = form.save(commit=False)
  user.save()

  send_activation_email.send(self, user)

  return HttpResponse('Confirm your email address.')
```

Słuchanie zdarzeń

Oczywiście ktoś musi słuchać naszego krzyczenia. Każdy framework definiuje to w nieco inny sposób, ale idea zazwyczaj jest taka sama: potrzebujemy *listenera*, który przetworzy nasze zdarzenia.

Słuchanie zdarzeń

```
class OrderShipped {
  public $order;

  public function __construct(Order $order) {
      $this->order = $order;
  }
}

class SendShipmentNotification {

    public function handle(OrderShipped $event) {
        $event->(...);
    }

}

// (...)

event(new OrderShipped($order));
```

Czego słuchać?

Wszystkiego, czego odpowiedzi tak naprawdę nie potrzebujemy.

Zastanówmy się jak wiele takich akcji realizowanych jest w naszych projektach? Wszystko, co związane z wysyłaniem mejli, zapisywaniem plików, bardzo często ze zmianami w bazie danych.

Kolejkowanie zdarzeń

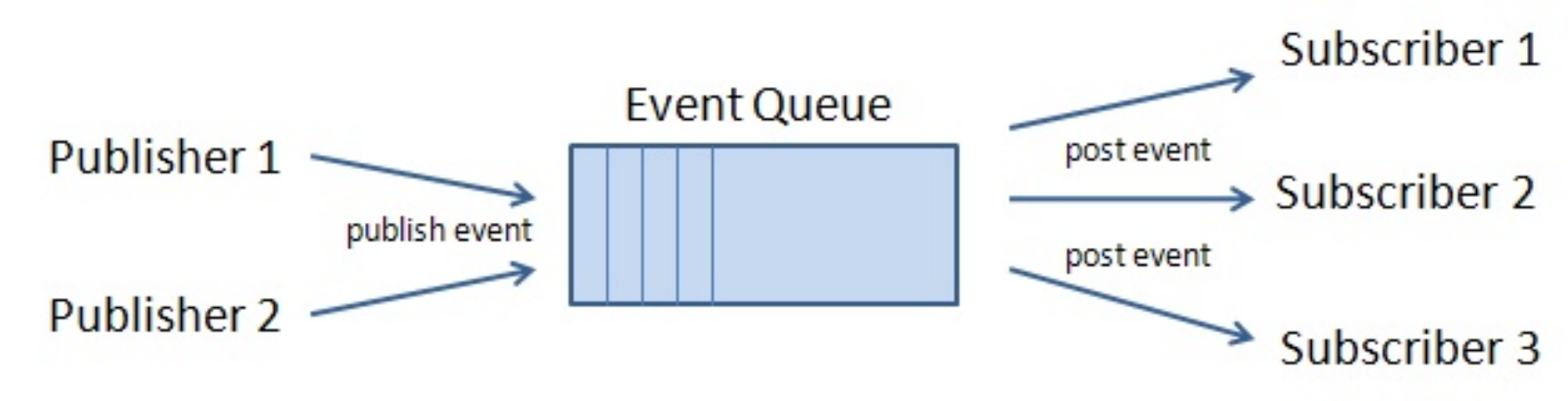
Kto pierwszy, ten lepszy?

Skoro wynik niektórych operacji jest dla nas czasami nieistotny i możemy realizację takiego zadania przerzucić na zdarzenie/słuchacza to może warto byłoby zastanowić się czy te zdarzenia muszą sie wykonać synchronicznie?

Kto pierwszy, ten lepszy?

Przyjmijmy, że wysyłanie mejla to skomplikowana sprawa. Trzeba się połączyć z serwerem poczty, wyciągnąć z bazy danych użytkownika, przerenderować wiadomość i oczywiście ją wysłać. Trwa to stosunkowo długo i czy faktycznie użytkownik musi czekać na koniec zadania żeby otrzymać powiadomienie *Odbierz mejla i aktywuj konto*?

Kolejkowanie zdarzeń



Rysunek 2: Idea kolejkowania zdarzeń

Kolejkowanie zdarzeń

Można utworzyć w dowolny sposób kolejkę zdarzeń do wykonania i następnie za jej pomocą zarządzać ich wykonywaniem.

Kolejkowanie zdarzeń

Korzystając z brokera wiadomości (RabbitMQ, Celery, Redis) można stworzyć kolejkę wewnątrz serwisu lub łącząc ze sobą (mikro)serwisy.

EDA

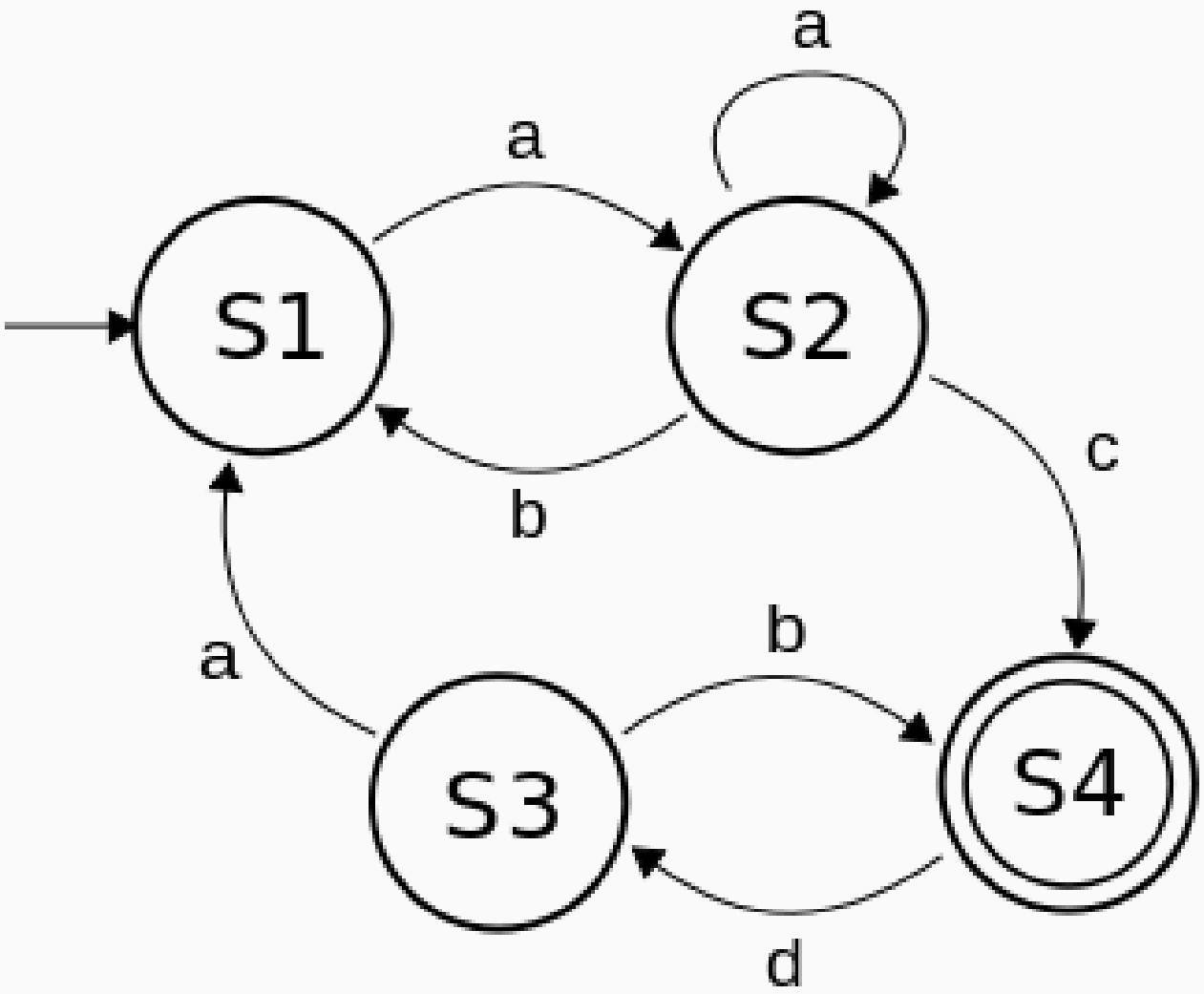
Event-driven architecture

Naturalnym rozwinięciem idei zdarzeń będzie architektura sterowana zdarzeniami, czyli *Event-driven architecture*, EDA.

Event-driven architecture

Najpierw definicje: **zdarzenie** powinno być interpretowane wówczas jako zmianę w stanie danego systemu.

Maszyna stanu



Event-driven architecture

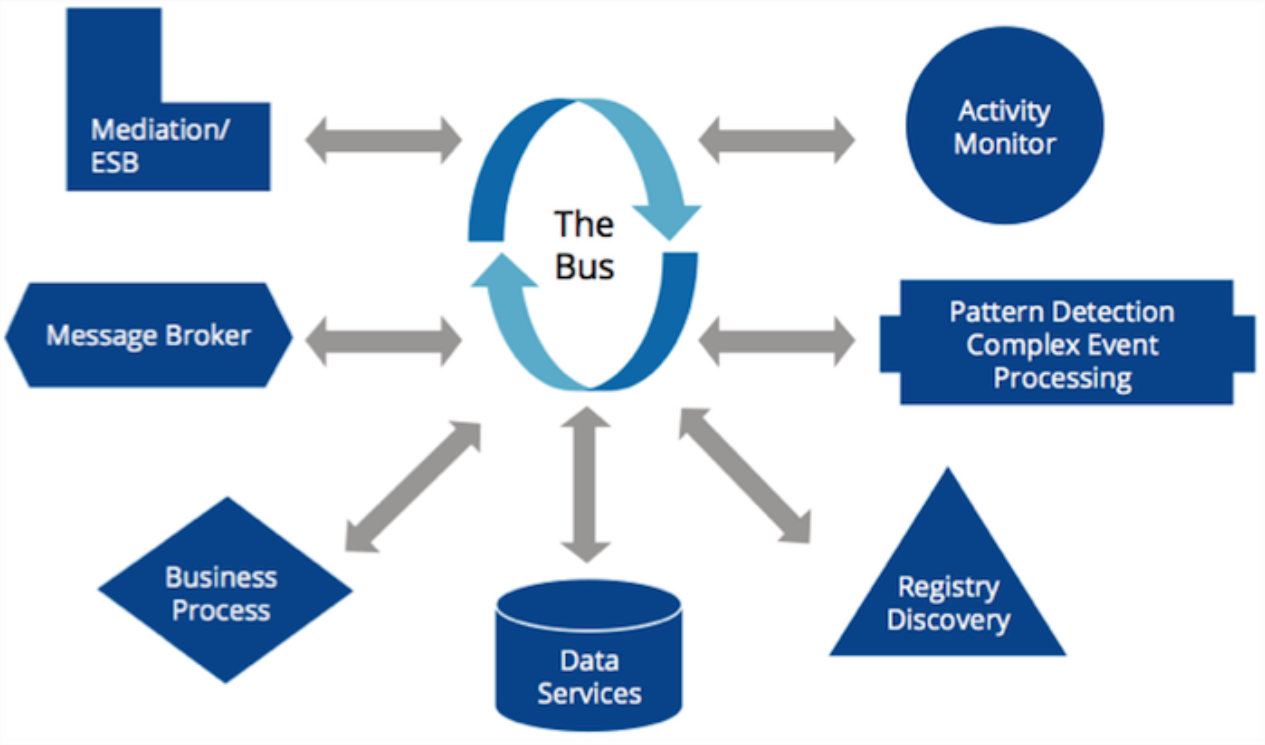
Przy EDA zdarzenia nie są klasami samymi w sobie, a bardziej ustandaryzowanymi komunikatami, które przesyłane są do konsumentów. Taki *event* najczęściej zawiera w sobie nazwę, timestamp, flagę typu zdarzenia oraz dodatkowe dane, które będą potrzebne przy przetwarzaniu zdarzenia.

Event-driven architecture

A więc: zmiena się stan użytkownika; przykładowo flaga `active` zmieniła swoją wartość z `true` na `false`. System powinien wyłapać taką zmianę i rozpropagować wiadomość informującą o tym zdarzeniu.

Każdy mikroserwis wchodzący w skład systemu odnotuje tę informację i zrobi to, co uzna za stosowne.

Event-driven architecture



Event-driven architecture

EDA może być potężnym narzędziem. Jest również coraz bardziej popularniejszym rozwiązaniem w projektach zarówno komercyjnych, jak i opensourcowych, więc warto się z nim zapoznać.

Podsumowanie

Bibliografia i ciekawe źródła

- https://en.wikipedia.org/wiki/Observer_pattern
- https://laravel.com/docs/5.6/eloquent#observers
- https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/integration-event-based-microservice-communications
- https://en.wikipedia.org/wiki/Event-driven_architecture
- https://brilliant.org/wiki/finite-state-machines/
- https://wso2.com/blogs/thesource/2016/05/how-you-can-increase-agility-and-expandability-with-event-dr

Pytania?

Kod prezentacji dostępny jest w repozytorium git pod adresem
`https://bitbucket.org/krewak/pwsz-ppsi2`

Wszystkie informacje dot. kursu dostępne są pod adresem
`http://pwsz.rewak.pl/kursy/6`