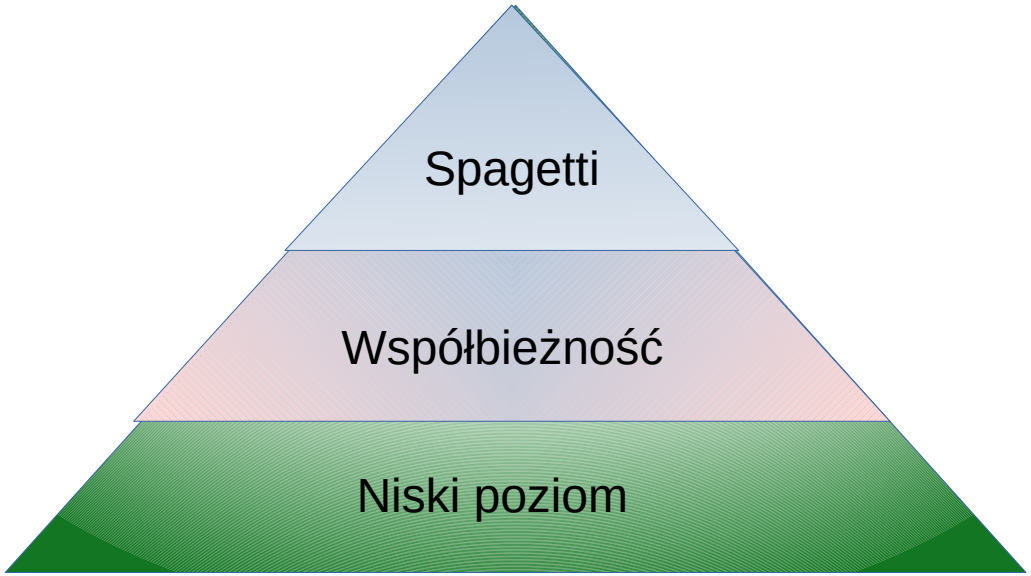
Wzorce i idiomy programowania dla systemów wbudowanych

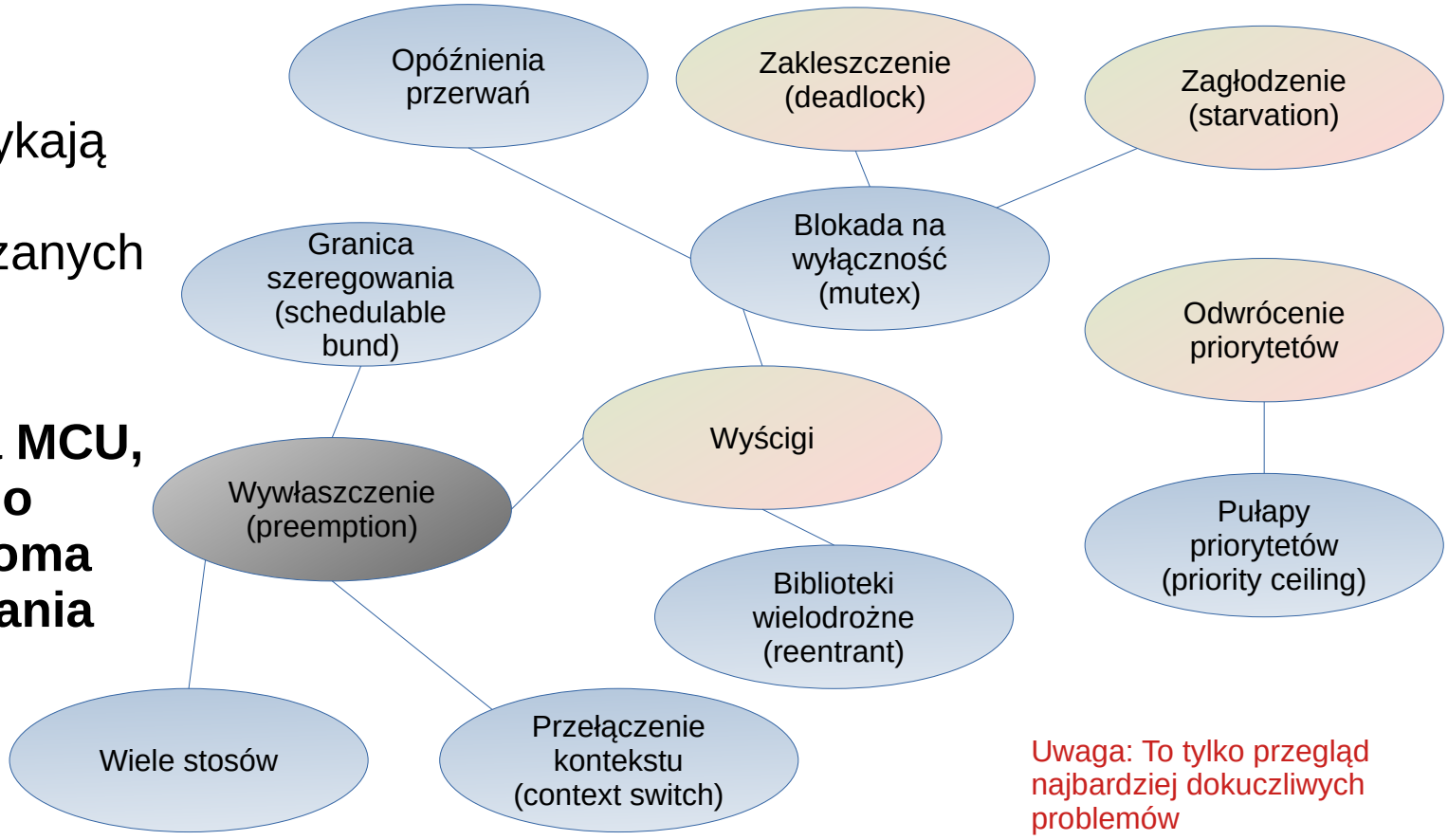
Piramida kodu...



- Kod na niskim poziomie, bezpośrednio operuje „na rejestrach”
- Programowanie współbieżne, tworzy problemy związane z współdzielonym dostępem do danych
- Kod „typu spagetti”, pojawia się w wyniku reaktywnego rozwiązywania problemów które „przydarzają się” w trakcie tworzenia rozwiązania

Podejście współbieżne

- Rozwiązania współbieżne borykają się z szeregiem problemów związanych z synchronizacją
- W zasadzie… **programując na MCU, zawsze mamy do czynienia z kilkoma wątkami wykonania** (przerwania)



Uwaga: To tylko przegląd najbardziej dokuczliwych problemów

„Superloop”

- Oprogramowanie pracuje w ciągłej pętli, przetwarzając po kolei zadania
- Brak możliwości przewidywania czasu wykonania każdego z zadań
- Możliwość zablokowania wykonania wszystkich zadań poprzez jedno z nich
- Brak determinizmu wykonania
- W praktyce… **być może dopuszczalne jako „wczesny prototyp”…**
- Ten wzorzec jest określany jako sekwencyjny
- Nie daje możliwości łatwego rozszerzania
- Jednostka centralna (CPU/MCU), ciągle wykonuje kod konsumując energię
- Szereg zadań ze względu na warunki, przez większość czasu … nie robi niczego

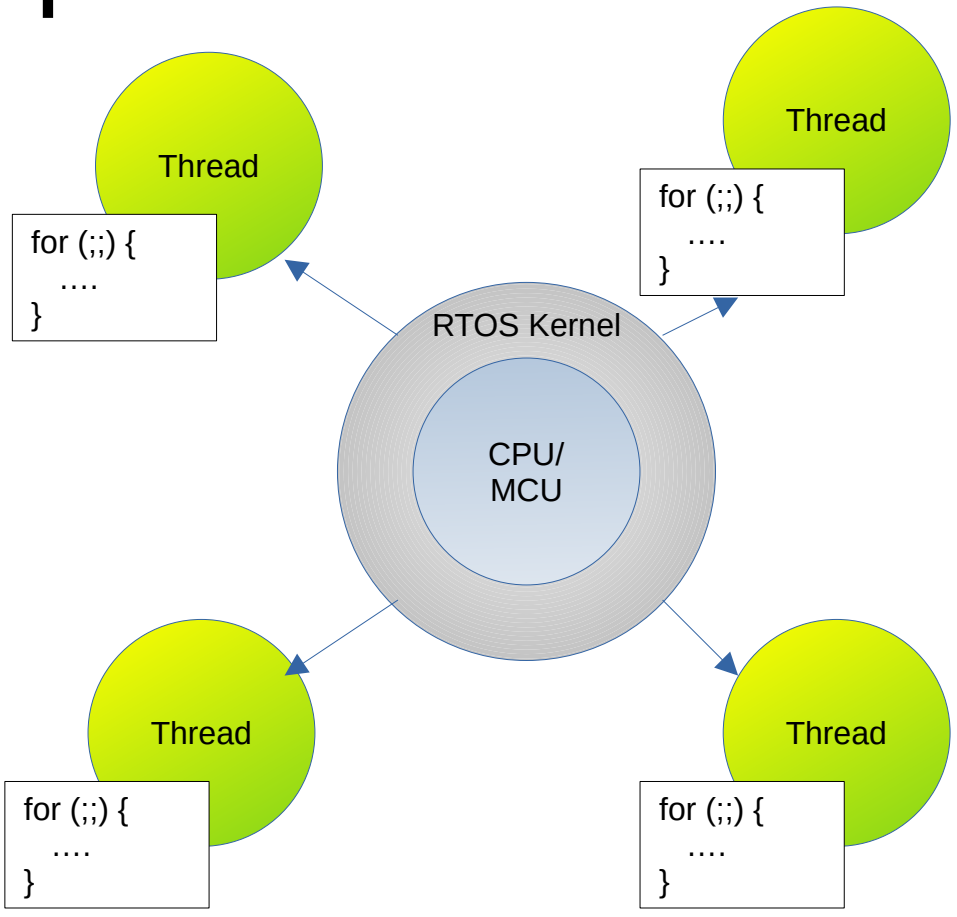
```
#include <….>
…
int main(void)
{
    …
    setup();
    …
    for (;;) {
        …
        task1();
        task2();
        …
    }
}
```

Dalej „superloop” ale „timery programowe”

- Każde z zadań sprawdza czy nastąpił czas jego wykonania
- W przypadku wykrycia momentu w czasie na działanie, wykonywany jest kod zadania. Jeśli ten moment nie nastąpił, zadanie jest natychmiast kończone
- Niejawnie zakłada międzyinnymi to że kod wykona się w czasie krótszym niż jednostka pomiaru czasu w przypadku każdego zadania
- Brak determinizmu aplikacji w przypadku przekroczenia reżimu czasu przez którekolwiek zadań
- Brak przewidzianej reakcji na „przegapiony zegar”
- W praktyce… **rozwiązania o ograniczonej funkcjonalności**
- Dalej brak możliwości rozszerzania aplikacji
- Ciągle program główny wykonuje „puste przebiegi” konsumując energię

Wiele „superloop” – RTOS

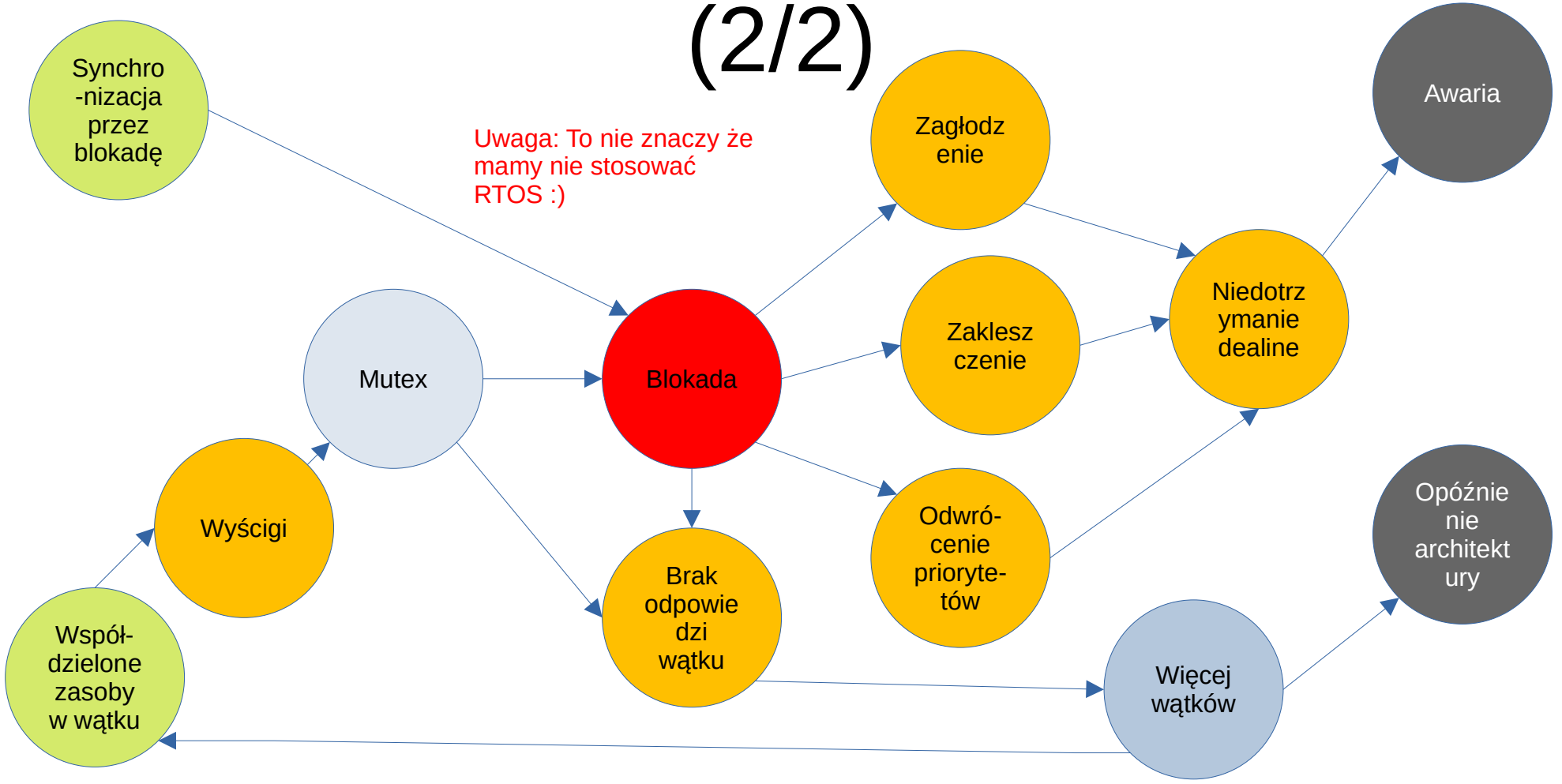
- Każde z zadań pracuje we własnej pętli
- Wywłaszczane jest przez mechanizmy planisty RTOS
- Aplikacje pracują (w miarę) sprawnie
- Szeregowanie w czasie uzależnione od mechanizmów wywłaszczenia
- W praktyce… **rozwiązanie często wybierane jako pierwsze po zastosowaniu RTOS**
- Kod jest łatwo rozszerzalny – z punktu widzenia twórcy pojedynczego Thread/Task, jest to samodzielna aplikacja
- System wymaga kreowania oddzielnych stosów dla każdego zadania
- Wykonuje puste przebiegi konsumując energię
- Przełączanie zadań, następuje z użyciem przerwania które przełącza kontekst wykonania



Konsekwencje zastosowania RTOS (1/2) (precyzyjniej.. planisty)

- Daje możliwość rozszerzania aplikacji o nowe zadania
- Utrzymuje iluzję programowania sekwencyjnego dla każdego z zadań
- W większym stopniu i w sposób bardziej kontrolowany używa zasobów obliczeniowych systemu
- Wątki dają się podzielić w domenie czasu (planista może faworyzować dane zadanie przydzielając mu więcej/częściej czas wykonania)
- W przypadku gdy wszystkie zadania nie wykonują produkcyjnego kodu, system może przechodzić do poziomu niższego poboru energii
- Mogą być stosowane techniki planowania wykonania zadań (RMS: https://pl.wikipedia.org/wiki/Rate-monotonic_scheduling)

Konsekwencje stosowania RTOS (2/2)



Problemy współbieżności (znowu)

- Wielu problemów współbieżności można uniknąć stosując komunikaty przesyłane do/pomiędzy wątkami a nie współdzieląc dane
- Każdy z wątków może posiadać własną kolejkę gdzie komunikaty będą procesowane
- Najlepiej przecież nie współdzielić… niczego

Dobre praktyki współczesnych architektur systemów wbudowanych

1) Izoluj dane. Mają być prywatne dla wątku.
   - Nie współdziel danych pomiędzy wątkami
2) Komunikacje pomiędzy wątkami (o ile konieczna), realizuj z użyciem asynchronicznych zdarzeń
   - Pozwalaj działać wątkom bez blokad wynikających z innych zadań
3) Organizuj wątek wokół „pompy wiadomości”
   - Blokady ograniczaj do obsługi kolejki komunikatów w głównej pętli wątku

Jasne że to wygląda na „dobre rady wujka”
za chwilę przyjrzymy się jak to osiągnąć...

Taki wzorzec nazywany jest:
Active Object

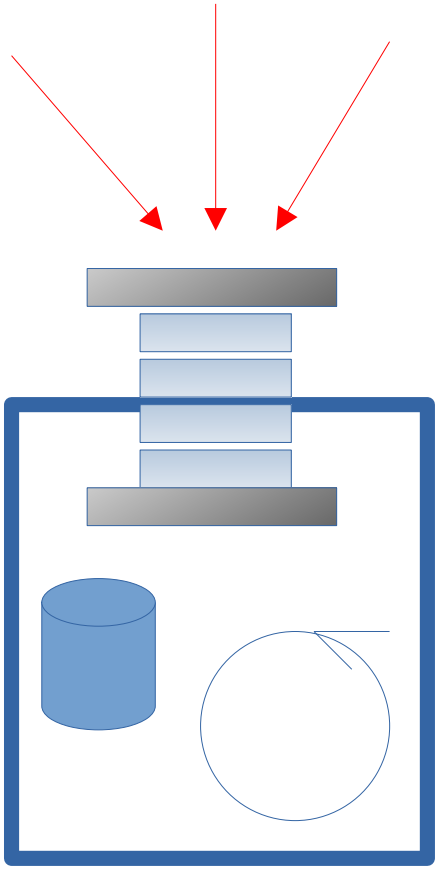
Pompa wiadomości

```
void thread(void * params) { // RTOS thread
    ActiveObject * self = (ActiveObject *) params;

    for (;;) { // thread superloop
        Event e; // message object
        // wait for any event and receive it
        QueueReceive(self->queue, &e, MAX_DELAY); // BLOCKING!!!

        // distpatch message to object
        ActiveObject_dispatch(self, &e); // NO BLOCKING!!!
    }
}
```

- Wątek obsługuje jedną wiadomość i wraca do pętli głównej (RTC – Run-To-Completion Event processing)
- Brak jawnego oczekiwania „w martwej pętli”
- Inversion of Control w porównaniu do „tradycyjnego podejścia z RTOS). Własny kod wątku decyduje kiedy ma być uruchomiony. Nawet w przypadku gdy otrzyma kontekst wykonania od planisty

Wzorzec Active Object

- Active Object =
  - prywatne dane + prywatny wątek + prywatna kolejka
- Jedyna komunikacja z AO, odbywa się przez zasilenie jego kolejki zdarzeń w sposób asynchroniczny
- Zdarzenie jest obsługiwane do zakończenia przez wątek
  - Brak mutexów innych niż wynikających z obsługi kolejki
- Nikt nie wymaga by to było OOP!
- AO jest abstrakcją wyższego rzędu nad RTOS
- RTOS jest dalej potrzebny do szeregowania procesów i dostarczania innych mechanizmów komunikacji/synchronizacji



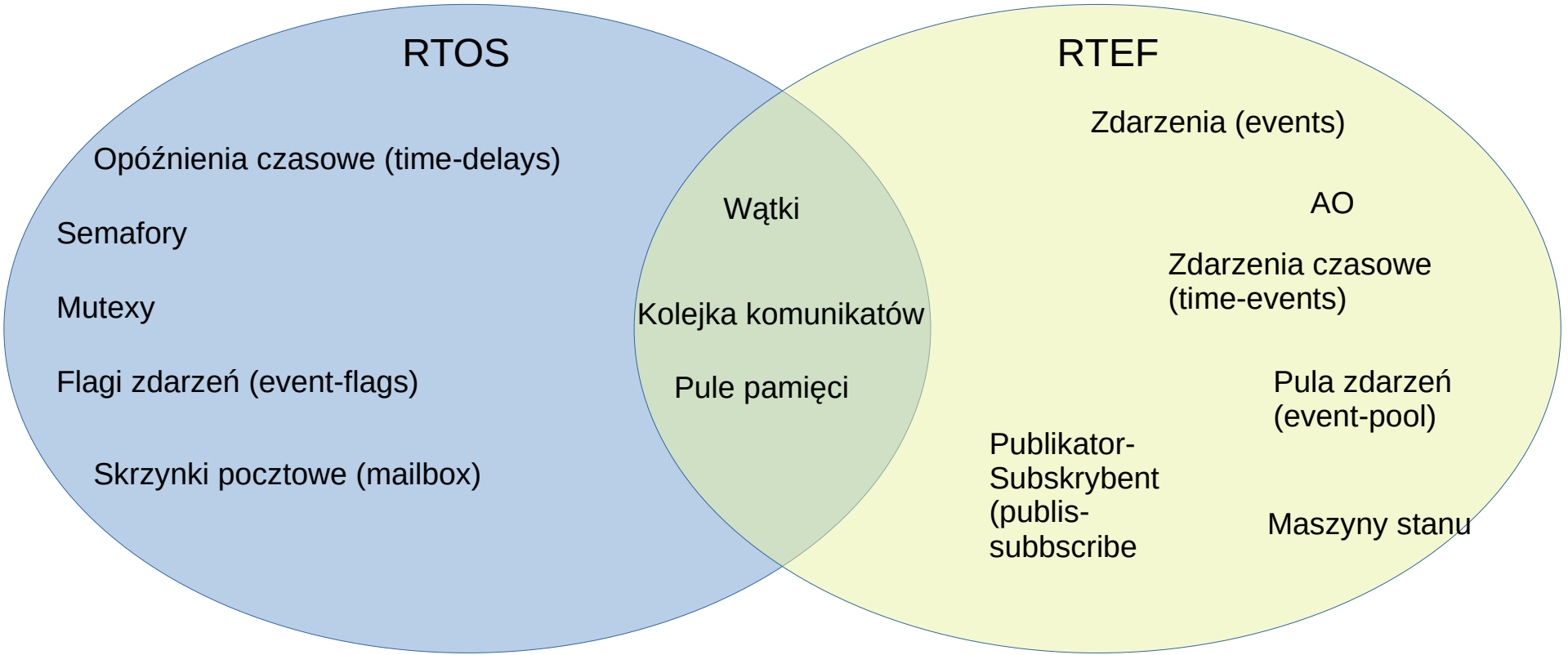
To nie jest „nowe podejście”!

- Carl Hewitt’s – idea Aktorów 1970
- Praca Real-Time Object Oriented Modeling (ROOM, 1994)
- UML Active Object / Active Class (UML-RT „Capsules”)

AO receptą kompletną?

- Czy AO rozwiązuje wszystkie problemy związane z blokowaniem?
- Do szeregowania dalej możemy używać RTOS
- Wymagane są jednak mechanizmy które będą wspierały podejście z architekturą komunikatów

Real-Time Framework (RTEF)



Komunikacja z użyciem AO