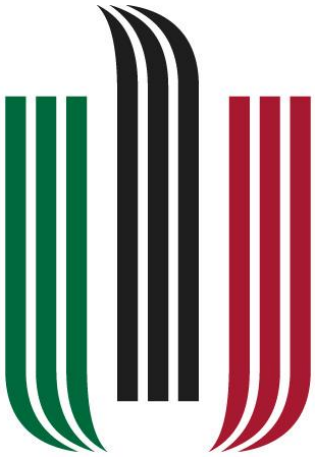
Laboratorium VII

Wzorce projektowe

dla programowania współbieżnego

Dominik Marek

18 listopada 2024



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Wyzwania dla projektantów sieciowych i współbieżnych aplikacji

1. Schmidt et al. wymieniają cztery istotne aspekty sieciowych i współbieżnych aplikacji:
    - Dostęp do usługi i konfiguracja
    - Obsługa zdarzeń
    - Współbieżność
    - Synchronizacja
2. **Dostęp do usługi i konfiguracja** (Service access and configuration): związany z tym, że komponenty sieciowej aplikacji współpracują przy pomocy różnych mechanizmów i protokołów komunikacji (np. IPC, TCP/IP, TELNET/FTP/HTTP, RPC). (Poza naszym zainteresowaniem.)
3. **Obsługa zdarzeń** (Event handling): związana z aplikacjami sterowanymi zdarzeniami (event-driven). Aplikacje takie wyróżniają się od aplikacji 'tradycyjnych' trzema charakterystycznymi cechami:

- Przetwarzanie w aplikacji jest wywoływane zdarzeniami, które występują asynchronicznie. Zdarzenia te mogą pochodzić od sterowników urządzeń, portów I/O, sensorów, klawiatury i myszy, sygnałów, etc.
- Większość zdarzeń powinna być obsłużona szybko (np. ze względu na czas odpowiedzi).
- Aplikacje sterowane zdarzeniami nie mają wpływu na kolejność, w której przychodzą zdarzenia. Dlatego może być konieczna kontrola i wykrywanie nielegalnych transakcji.

4. **Współbieżność** (concurrency) dotyczy mozliwosci wykonywania wielu zadań jednocześnie przez wiele procesów/wątków, np. jednoczesnej obsługi żądań wielu klientów przez serwer. Użycie wątków może poprawić responsywność, efektywność i strukturę (design) aplikacji.
5. **Synchronizacja** (synchronization): programowanie współbieżne jest trudniejsze niż sekwencyjne ze względu na konieczność synchronizacji dostępu do zasobów dzielonych.

Wzorce współbieżności (Concurrency patterns)

1. Aktywny obiekt (Active object)
2. Monitor object
3. Half-Sync / Half-Async
4. Leader-Follower
5. Thread-Specific Storage

Wzorzec Active Object

Aktywny obiekt *oddziela wykonanie metody od wywołania metody* aby poprawić współbieżność i uprościć synchronizowany dostęp do obiektów, które są umieszczone w swoich własnych wątkach.

Alternatywna nazwa

Concurrent object

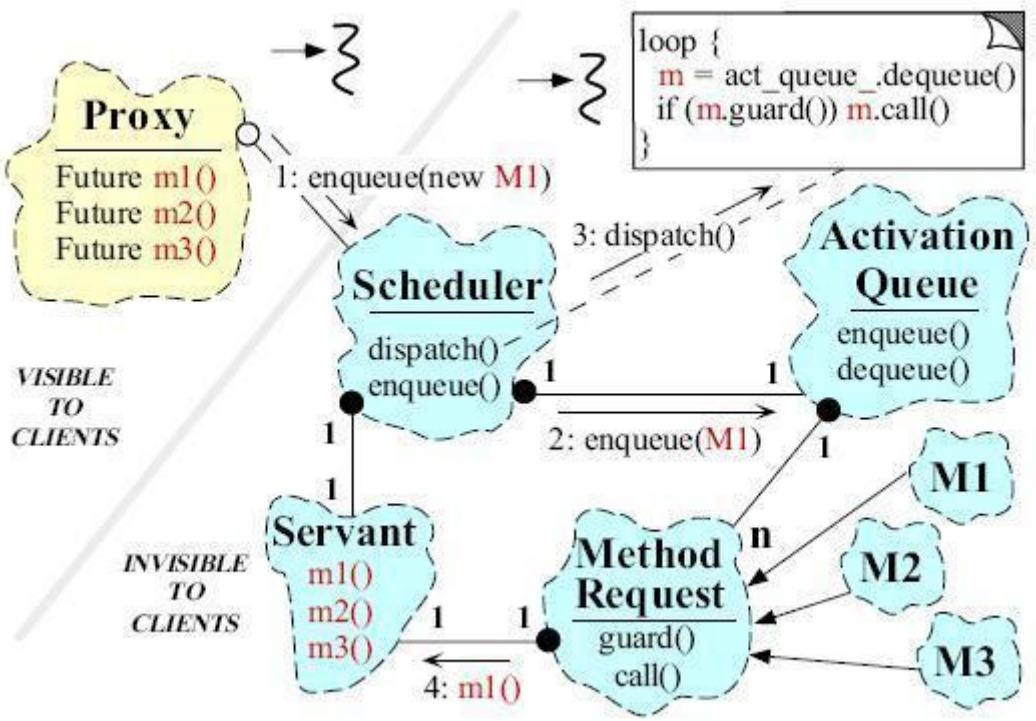
Kontekst

Klienci, którzy odwołują się do obiektów wykonujących się w osobnych wątkach.

Rozwiązanie

Oddzielenie wywołania metody od jej wykonania. Wywołanie metody odbywa się w wątku klienta, lecz jej wykonanie w osobnym wątku - pracownika. Powinno to być tak zaprojektowane, żeby z punktu widzenia klienta wyglądało to na zwykłe wywołanie metody.

Struktura



Active object składa się z sześciu składników:

1. **Proxy** (Pośrednik). Reprezentuje interfejs obiektu. Udostępnia interfejs dla klientów, przy pomocy którego wywołują oni publicznie dostępne metody aktywnego obiektu. Wywołanie takiej metody powoduje utworzenie obiektu *Method Request* i umieszczenie go w kolejce *Activation Queue* należącej do *Schedulera*.

2. **Servant** (Pracownik). Dostarcza implementacji obiektu. Implementuje metody zdefiniowane w *Proxy* i odpowiednim *Method Request*. Metoda *Servanta* jest wywołana gdy odpowiadające jej żądanie metody (*Method Request*) jest wykonane przez *Scheduler*. *Servant* wykonuje się w wątku *Schedulera*. *Servant* może dostarczać dodatkowych metod, które mogą posłużyć do implementacji strażników w *Method Request*

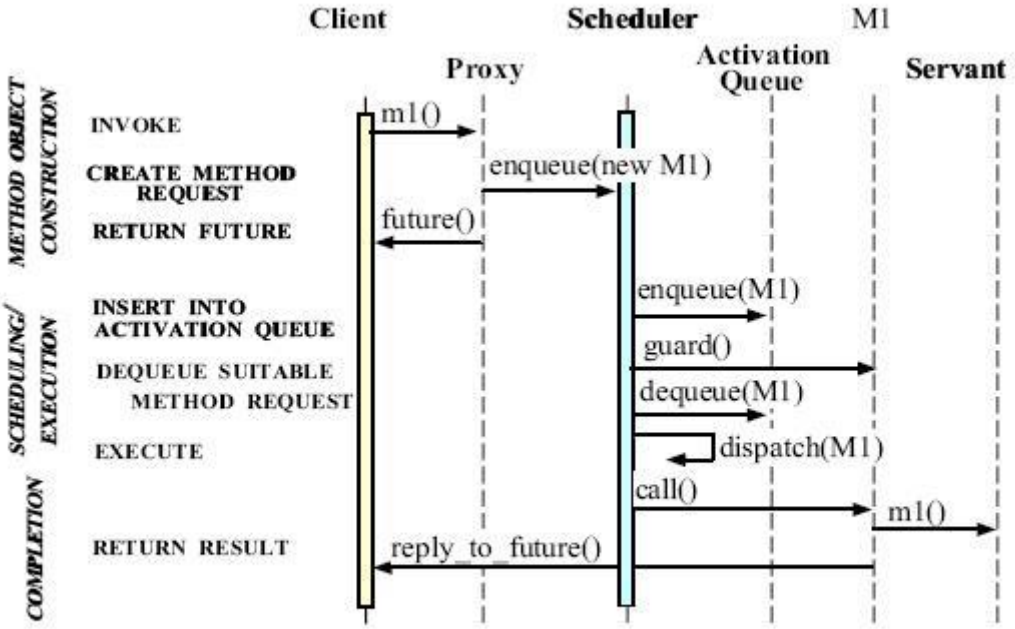
3. **Method Request** (Żądanie metody). Używane jest do przekazanie *kontekstu wywołania* metody (np. parametrów) z Proxy do Schedulera uruchomionego w osobnym wątku. Abstrakcyjna klasa *Method Request* definiuje interfejs dla wykonywania metod Aktywnego Obiektu. Interfejs ten zawiera również metody *strażników* (guard), które są mogą być użyte do sprawdzenia, czy warunki związane z synchronizacją są spełnione.

4. **Activation Queue** (Kolejka aktywacji). Zawiera bufor oczekujących *Method Request* utworzonych przez *Proxy*. Jest zasobem dzielonym dla wątków klienta i pracownika - pierwszy jest producentem żądań metody, drugi ich konsumentem (przez Scheduler).

5. **Scheduler** (Zarządca). Wykonuje się w osobnym wątku, zarządzając *Activation Queue*. Decyduje, które *Method Request* ma być zdjęte z kolejki i wykonane w *Servant* implementującym metodę. Ta decyzja oparta jest o różne kryteria, np. kolejność umieszczenia metod w kolejce, spełnienie pewnych warunków związanych z synchronizacją (np. zwolnienie miejsca w buforze). Uwarunkowania związane z synchronizacją są zwykle sprawdzane przy użyciu metod - *strażników*.

6. **Future** (Zmienna terminowa). Pozwala klientowi pobranie wyniku wykonania metody, gdy *Servant* zakończy jej wykonanie. Po wywołaniu metody w *Proxy*, obiekt Future jest od razu zwracany. Future rezerwuje miejsce na wyniki wywołania metody. Gdy klient chce te wyniki pobrać, albo się blokuje do czasu pojawienia się wyników, albo periodycznie sprawdza przez *polling*.

Przebieg wywołania



Zadania:

Zaimplementować bufor jako aktywny obiekt (Producenci-Konsumenci)

Wskazówki:

1. Pracownik powinien implementować samą kolejkę (bufor) oraz dodatkowe metody (czyPusty etc.), które pomogą w implementacji strażników. W klasie tej powinna być tylko funkcjonalność, ale nie logika związana z synchronizacją.
2. Dla każdej metody aktywnego obiektu powinna być specjalizacja klasy MethodRequest. W tej klasie m.in. zaimplementowana jest metoda guard(), która oblicza spełnienie warunków synchronizacji (korzystając z metod dostarczonych przez Pracownika).
3. Proxy wykonuje się w wątku klienta, który wywołuje metodę. Tworzenie Method request i kolejkowanie jej w Activation queue odbywa się również w wątku klienta. Servant i Scheduler wykonują się w osobnym (oba w tym samym) wątku.

1.Rozwiązania

1. ActiveObject

ActiveObject pełni rolę centralnego komponentu aplikacji, odpowiedzialnego za zarządzanie aktywnym obiektem. Składa się z trzech głównych elementów: bufora (*Buffer*), scheduler'a (*Scheduler*) i *Proxy*. Bufor przechowuje dane, które konsumenci mogą pobierać, a producenci dodawać, jednak rzeczywiste operacje są synchronizowane za pomocą schematu aktywnego obiektu. Scheduler działa jako osobny wątek, przetwarzający asynchroniczne żądania. Klasa udostępnia metodę *getProxy()*, pozwalającą klientom uzyskać referencję do interfejsu Proxy.

2. Buffer

Buffer jest strukturą danych opartą na kolejce, przechowującą obiekty przesyłane między producentami a konsumentami. Zawiera metody *add()* i *remove()* do odpowiednio dodawania oraz usuwania elementów, a także pomocnicze metody *isFull()* i *isEmpty(),* które sprawdzają aktualny stan bufora. Klasa nie zajmuje się synchronizacją wielowątkową, delegując tę odpowiedzialność do wyższych poziomów (logika strażników w *IMethodRequest*). Dzięki temu klasa *Buffer* jest uproszczona i skoncentrowana na podstawowej funkcjonalności.

3. CustomFuture

CustomFuture to mechanizm umożliwiający realizację operacji asynchronicznych. Klasa przechowuje obiekt, który jest wynikiem wykonania operacji, i udostępnia metody pozwalające na jego ustawienie (*setObject*) oraz pobranie (*getObject*). Dodatkowo posiada metodę *isReady()*, która sprawdza, czy wynik jest dostępny. Konsumenci mogą czekać na zakończenie operacji w sposób nieblokujący, regularnie sprawdzając stan *CustomFuture.*

4. IMethodRequest oraz klasy dziedziczące: AddRequest i RemoveRequest

IMethodRequest jest interfejsem, który definiuje strukturę żądań (*method requests*) w aktywnym obiekcie. Każda metoda aktywnego obiektu ma swoją implementację w formie klasy. *AddRequest* odpowiada za dodawanie obiektów do bufora i wykonuje tę operację tylko, gdy bufor nie jest pełny (sprawdzane przez metodę *guard()*). Analogicznie *RemoveRequest* umożliwia usuwanie elementów z bufora, ale tylko wtedy, gdy ten nie jest pusty. Klasy te implementują metodę *call(),* realizującą właściwe operacje, oraz *guard(),* określającą warunki synchronizacji.

5. Scheduler

Scheduler zarządza kolejką aktywacyjną, czyli listą żądań do wykonania. Działa w osobnym wątku, przetwarzając żądania w pętli. Pobiera pierwsze żądanie z kolejki, a jeśli warunki synchronizacji (sprawdzane przez *guard()*) są spełnione, wywołuje metodę *call().* W

przeciwnym razie żądanie jest odkładane z powrotem na koniec kolejki. Dzięki temu Scheduler zapewnia płynne i asynchroniczne przetwarzanie żądań producentów i konsumentów.

6. Servant

Servant implementuje interfejs *Proxy,* pełniąc rolę pośrednika między klientami a aktywnym obiektem. Udostępnia metody *add()* i *remove(),* które tworzą odpowiednie żądania (*AddRequest, RemoveRequest*) i dodają je do kolejki aktywacji w Scheduler. Operacje są wykonywane asynchronicznie przez Scheduler, co zapewnia izolację logiki klienta od rzeczywistego stanu bufora.

7. Consumer

Consumer to klasa reprezentująca konsumenta działającego w osobnym wątku. Konsument cyklicznie wywołuje metodę *proxy.remove(),* aby pobrać element z bufora. Dopóki *CustomFuture* nie jest gotowy (*isReady()*), konsument oczekuje, wyświetlając komunikaty o oczekiwaniu. Po pobraniu elementu konsumuje go (wyświetla komunikat) i wprowadza opóźnienie, symulując czas potrzebny na przetwarzanie.

8. Producer

Producer to klasa reprezentująca producenta, który również działa w osobnym wątku. W pętli generuje losowe liczby i dodaje je do bufora za pomocą *proxy.add().* Podobnie jak konsument, producent wprowadza opóźnienie między kolejnymi operacjami, co pozwala na symulację rzeczywistego procesu produkcji danych. Producent działa niezależnie od stanu konsumentów, co umożliwia pełną asynchroniczność systemu.

9. Main

Klasa Main odpowiada za uruchomienie całego programu. Tworzy instancję *ActiveObject,* który zarządza współpracą między producentami i konsumentami za pomocą bufora aktywnego obiektu. Następnie inicjalizuje tablice wątków producentów (*Producer*) i konsumentów (*Consumer*), przydzielając im identyfikatory i referencję do Proxy. Wątki są uruchamiane i odpowiednio zsynchronizowane poprzez metody *start()* oraz *join().* Klasa demonstruje działanie systemu w kontrolowanym środowisku wielowątkowym.

Podsumowanie

Ten system, oparty na wzorcu aktywnego obiektu, pozwala na efektywną i bezpieczną współpracę wielu wątków w środowisku wielowątkowym. Synchronizacja operacji odbywa się dzięki logice strażników (*guard()*) i kolejce aktywacyjnej (*Scheduler*), co pozwala oddzielić logikę przetwarzania od rzeczywistego stanu bufora.

```java
public class ActiveObject {

    private Buffer buffer;
    private Scheduler scheduler;
    private Proxy proxy;


    public ActiveObject(int queueSize) {
        this.buffer =  new Buffer(queueSize);
        this.scheduler = new Scheduler();
        this.proxy = new Servant(this.buffer, this.scheduler);
        scheduler.start();
    }


    public Proxy getProxy(){
        return proxy;
    }

}
```

```java
import java.util.LinkedList;
import java.util.Queue;

public class Buffer {


    private final int buffSiz;
    private Queue<Object> buffer;

    public Buffer(int buffSiz){
        this.buffSiz = buffSiz;
        this.buffer = new LinkedList<Object>();
    }


    public void add(Object o){
        if (!this.isFull()){
            this.buffer.add(o);
        }
    }


    public Object remove(){
        if(this.isEmpty()){
            return null;
        }
        return this.buffer.remove();
    }



    public  boolean isFull(){
        return buffer.size() == buffSiz;
    }

    public boolean isEmpty(){
        return buffer.isEmpty();
    }
```

```java
public class CustomFuture {

    private Object object;

    public void setObject(Object object) {
        this.object = object;
    }

    public Object getObject() {
        return object;
    }

    public boolean isReady(){
        return object != null;
    }
}
```

```java
public interface IMethodRequest {

    boolean guard();
    void call();
}
```

```java
public class AddRequest implements IMethodRequest {


    private final Buffer buffer;
    private final Object object;


    public AddRequest(Buffer buffer, Object object) {
        this.buffer = buffer;
        this.object = object;
    }

    @Override
    public void call(){
        this.buffer.add(this.object);
    }


    @Override
    public boolean guard(){
        return !this.buffer.isEmpty();
    }

}
```

```java
public class RemoveRequest implements IMethodRequest{

    private Buffer buffer;
    private CustomFuture customFuture;


    public RemoveRequest(Buffer buffer, CustomFuture customFuture) {
        this.buffer = buffer;
        this.customFuture = customFuture;
    }
```

```
    @Override
    public void call(){
        customFuture.setObject(buffer.remove());
    }


    @Override
    public boolean guard(){
        return !this.buffer.isEmpty();
    }


}
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

public class Scheduler extends Thread {

    private Queue<IMethodRequest> activationQueue;


    public Scheduler() {
        activationQueue = new ConcurrentLinkedQueue<IMethodRequest>();
    }


    public void enqueue(IMethodRequest request) {
        activationQueue.add(request);
    }



    public void run() {
        while(true){
            IMethodRequest request = activationQueue.poll();
            if(request != null){
                if(request.guard()){
                    request.call();
                }else {
                    activationQueue.add(request);
                }
            }
        }
    }



}
```

```java
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

public class Scheduler extends Thread {

    private Queue<IMethodRequest> activationQueue;


    public Scheduler() {
        activationQueue = new ConcurrentLinkedQueue<IMethodRequest>();
    }


    public void enqueue(IMethodRequest request) {
        activationQueue.add(request);
    }



    public void run() {
        while(true){
            IMethodRequest request = activationQueue.poll();
            if(request != null){
                if(request.guard()){
                    request.call();
                }else {
                    activationQueue.add(request);
                }
            }
        }
    }



}
```

```java
public interface Proxy {

    void add(Object object);

    CustomFuture remove();
}
```

```java
public class Servant  implements Proxy{

    Buffer buffer;
    Scheduler scheduler;

    public Servant(Buffer buffer, Scheduler scheduler) {
        this.buffer = buffer;
        this.scheduler = scheduler;
    }


    @Override
    public void add(Object o){
        this.scheduler.enqueue(new AddRequest(this.buffer, o));
```

```
    }

    @Override
    public CustomFuture remove(){
        CustomFuture customFuture = new CustomFuture();
        this.scheduler.enqueue(new RemoveRequest(this.buffer,
customFuture));
    return customFuture;
    }


}
```

```
public class Consumer extends Thread{
    private int ID;
    private Proxy proxy;


    public Consumer(int ID, Proxy proxy){
        this.ID=ID;
        this.proxy = proxy;
    }


    @Override
    public void run(){

        int i = 0;
        while(i < 10){
            CustomFuture consumed =  proxy.remove();

            while(!consumed.isReady()){

                System.out.println("Consumer " +this.ID+ " is waiting");

                try {
                    Thread.sleep(400);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }

            System.out.println("Consumer " +this.ID+ " has eaten: " +
consumed.getObject());

            try {
                Thread.sleep(400);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            i++;
        }

    }
}
```

```java
import java.util.Random;

public class Producer extends Thread{

    private int ID;
    private Proxy proxy;

    private Random rand;


    public Producer(int ID, Proxy proxy) {
        this.ID = ID;
        this.proxy = proxy;
        rand = new Random();
    }


    @Override
    public void run(){

        int i = 0;
        while(i < 10){
             int tmp = this.rand.nextInt(100);

             proxy.add(tmp);

            System.out.println("Producer " + ID + " added:   " + tmp);

            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            i++;
        }
    }

}
```

```java
public class Main {
    private final static int PROD_COUNT = 5;
    private final static int CONS_COUNT = 3;
    private final static int ACTIVE_OBJECTS = 10;

    public static void main(String[] argv) throws InterruptedException {
        ActiveObject activeObject = new ActiveObject(ACTIVE_OBJECTS);
        Proxy proxy = activeObject.getProxy();
        Producer[] producers = new Producer[PROD_COUNT];
        Consumer[] consumers = new Consumer[CONS_COUNT];

        for (int i = 0; i < PROD_COUNT; i++) {
            producers[i] = new Producer(i, proxy);
        }

        for (int i = 0; i < CONS_COUNT; i++) {
            consumers[i] = new Consumer(i, proxy);
        }

        for (int i = 0; i < CONS_COUNT; i++) {
```

```
            consumers[i].start();
        }

        for (int i = 0; i < PROD_COUNT; i++) {
            producers[i].start();
        }

        for (int i = 0; i < PROD_COUNT; i++) {
            producers[i].join();
        }

        for (int i = 0; i < CONS_COUNT; i++) {
            consumers[i].join();
        }
    }
}
```

3.Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- http://brinch-hansen.net/papers/1999b.pdf ~ Per Brinch Hansen
- https://www.artima.com/insidejvm/ed2/threadsynch.html ~Bill Venners, Inside the Java Virtual Machine - Charter 20 Thread Synchronization
- https://docs.oracle.com/en/java/javase/19/index.html
- https://www.dre.vanderbilt.edu/~schmidt/PDF/Active-Objects.pdf
- https://en.wikipedia.org/wiki/Active_object