Programowanie współbieżne
Laboratorium nr 15

1 Framework fork / join

Framework fork / join [9] jest implementacją ExecutorService mająca na celu wydajne wykorzystanie wielu procesorów. Został zaprojektowany do obsługi zadań, które mogą być podzielone rekurencyjnie na części. Wykorzystuje on algorytm ang. work-stealing. Wątki, które nie mają co robić mogą podkradać zadania od wątków, które są zajęte.

Pseudokod [9]:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

Wybrane klasy związane z tym framework'iem:

- ForkJoinPool – egzekutor wykorzystywany do uruchamiania ForkJoinTask jak również „zwykłych" zadań, wykorzystuje algorytm work-stealing polegający na tym, że wątki będące w puli próbują znaleźć i wykonać zadania zgłoszone do puli lub „podkraść" je od innych wątków, które są zajęte,
- ForkJoinWorkerThread – wątek zarządzany przez egzekutor ForkJoinPool, który wykonuje zadania ForkJoinTask,
- ForkJoinTask<V> – abstrakcyjna klasa bazowa dla zadań uruchamianych przez ForkJoinTask, istnieją następujące klasy pochodne:
  - CountedCompleter,
  - RecursiveAction,
  - RecursiveTask.

Metody służące do wykonywanie zadań:

	Wywołania pochodzące od klientów „non-fork/join"	Wywołania pochodzące „z wewnątrz" obliczeń „fork/join"
Zarządź asynchroniczne wywołanie	ForkJoinPool.execute(ForkJoinTask)	ForkJoinTask.fork()
Poczekaj na rezultat	ForkJoinPool.invoke(ForkJoinTask)	ForkJoinTask.invoke()
Zarządź wykonanie i otrzymaj obiekt Future	ForkJoinPool.submit(ForkJoinTask)	ForkJoinTask.fork() (obiekty ForkJoinTasks są obiektami Future)

Przykład dotyczący wyznaczania maksymalnej wartości z elementów listy:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class FindMaxRecursiveTask extends RecursiveTask<Integer> {

    private List<Integer> input;

    public FindMaxRecursiveTask(List<Integer> input) {
        this.input = input;
    }

    @Override
```

```
    public String toString() {
        return super.toString() + " " + input;
    }

    @Override
    protected Integer compute() {

        if (input.size() <= 2) {
            return findMax(input);
        } else {
            List<FindMaxRecursiveTask> subtasks = createSubtasks();
            for (FindMaxRecursiveTask subtask : subtasks)
                subtask.fork();

            List<Integer> output = new ArrayList<>();

            for (FindMaxRecursiveTask subtask : subtasks)
                output.add(subtask.join());

            return findMax(output);
        }
    }

    private List<FindMaxRecursiveTask> createSubtasks() {
        List<FindMaxRecursiveTask> subtasks = new ArrayList<>();
        int length = input.size();
        int halfLength = length / 2;

        List<Integer> firstHalfList = input.subList(0, halfLength);
        FindMaxRecursiveTask firstSubtask = new FindMaxRecursiveTask(firstHalfList);
        System.out.println(this + " creatting subtask " + firstSubtask);
        subtasks.add(firstSubtask);

        List<Integer> secondtHalfList = input.subList(halfLength, length);
        FindMaxRecursiveTask secondSubtask = new FindMaxRecursiveTask(secondtHalfList);
        System.out.println(this + " creatting subtask " + secondSubtask);
        subtasks.add(secondSubtask);

        return subtasks;
    }

    private Integer findMax(List<Integer> list) {
        if (list.size() == 1)
            return list.get(0);
        if (list.size() == 2) {
            if (list.get(0) > list.get(1))
                return list.get(0);
            else
                return list.get(1);
        }
        throw new RuntimeException("List length error");
    }
}
```

```
import java.util.Arrays;
import java.util.concurrent.ForkJoinPool;

public class TestJoinFork {

    public static void main(String[] args) {
```

```
        ForkJoinPool forkJoin = new ForkJoinPool();
        Integer max = forkJoin.invoke(new FindMaxRecursiveTask(Arrays.asList(9, 98, 8, 87, 7, 765,↩
             654, 54321, 3210)));
        System.out.println("MAX = " + max);
    }

}
```

Wynik:

```
FindMaxRecursiveTask@6f753c95 [9, 98, 8, 87, 7, 765, 654, 54321, 3210] creatting subtask ↩
     FindMaxRecursiveTask@61b2df0b [9, 98, 8, 87]
FindMaxRecursiveTask@6f753c95 [9, 98, 8, 87, 7, 765, 654, 54321, 3210] creatting subtask ↩
     FindMaxRecursiveTask@658d57e5 [7, 765, 654, 54321, 3210]
FindMaxRecursiveTask@61b2df0b [9, 98, 8, 87] creatting subtask FindMaxRecursiveTask@208fd530 [9, 98]
FindMaxRecursiveTask@658d57e5 [7, 765, 654, 54321, 3210] creatting subtask FindMaxRecursiveTask@5268543e [7, 765]
FindMaxRecursiveTask@61b2df0b [9, 98, 8, 87] creatting subtask FindMaxRecursiveTask@6658d420 [8, 87]
FindMaxRecursiveTask@658d57e5 [7, 765, 654, 54321, 3210] creatting subtask FindMaxRecursiveTask@5cafb003 [654, ↩
     54321, 3210]
FindMaxRecursiveTask@5cafb003 [654, 54321, 3210] creatting subtask FindMaxRecursiveTask@75f5a224 [654]
FindMaxRecursiveTask@5cafb003 [654, 54321, 3210] creatting subtask FindMaxRecursiveTask@7b3252ea [54321, 3210]
MAX = 54321
```

2 Synchronizacja nieblokująca

„Wiele najnowszych badań [1] nad algorytmami współbieżnymi skupiało się na algorytmach nieblokujących, które wykorzystywały niepodzielne, niskopoziomowe instrukcje maszynowe, na przykład porównaj i zamień, zamiast blokad, by zapewnić spójność danych w trakcie współbieżnego dostępu."

„Od Javy 5.0 [1] można tworzyć w tym języku wydajne algorytmy nieblokujące, wykorzystując klasy zmiennych niepodzielnych, na przykład AtomicInteger lub AtomicReference." Wykorzystują one instrukcje CAS (ang. compare-and-swap).

„Algorytm nazywamy nieblokującym [1], jeżeli błąd lub zawieszenie dowolnego wątku nie powoduje błędu ani zawieszenia innego wątku. Algorytm nazywa się wolnym od blokad, jeżeli w każdym kroku jakiś wątek czyni postępy. Algorytmy, które używają wyłącznie CAS do koordynacji między wątkami, mogą jeśli zostaną należycie skonstruowane, być jednocześnie nieblokujące i wolne od blokad."

Opisy algorytmów nieblokujących pochodzą z:

1. Java theory and practice: Introduction to nonblocking algorithms. Look Ma, no locks! Brian Goetz, http://www.ibm.com/developerworks/java/library/j-jtp04186/

2. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D., Java Współbieżność dla praktyków, Helion 2007

Algorytm dotyczący nieblokującego stosu został opisany w: R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986

Algorytm dotyczący nieblokującej kolejki został opisany w: Śimple, Fast, and Practical Non-Blocking and Blocking Concurrent Queues"Maged M. Michael and Michael L. Scott, Symposium on Principles of Distributed Computing, 1996

2.1 Licznik

Blokujący licznik:

```
public final class Counter {
    private long value = 0;

    public synchronized long getValue() {
        return value;
    }
```

```
    public synchronized long increment() {
        return ++value;
    }
}
```

Nieblokujący licznik:

```
public class NonblockingCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        }
         while (!value.compareAndSet(v, v + 1));
        return v + 1;
    }
}
```

2.2 Nieblokujący stos

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = head.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead,newHead));
        return oldHead.item;
    }

    static class Node<E> {
        final E item;
        Node<E> next;

        public Node(E item) { this.item = item; }
    }
}
```

2.3 Nieblokująca kolejka

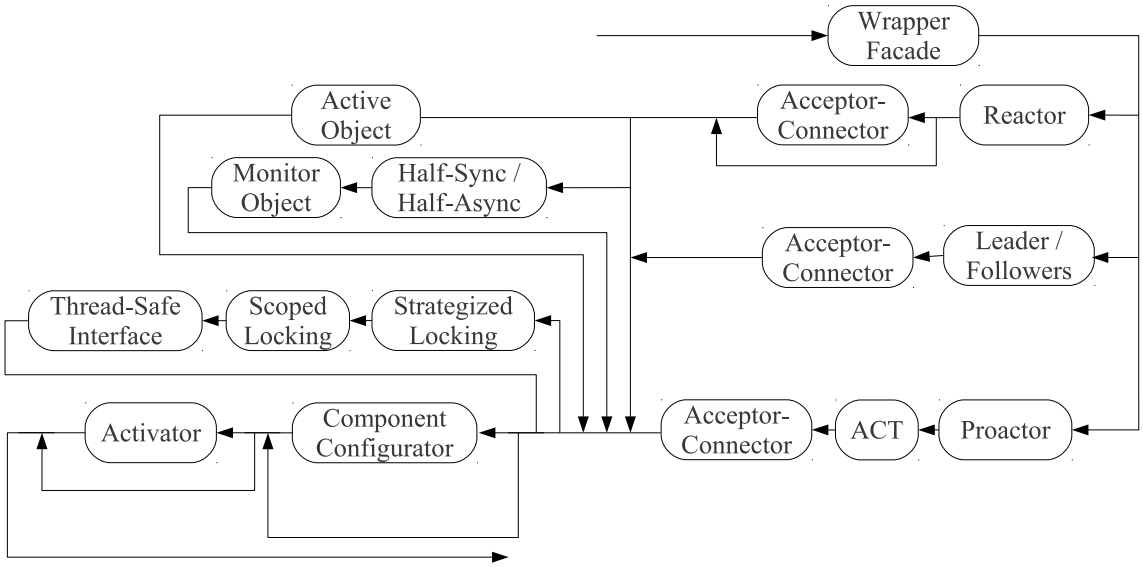
```
public class LinkedQueue <E> {
  private static class Node <E> {
      final E item;
      final AtomicReference<Node<E>> next;

      Node(E item, Node<E> next) {
          this.item = item;
          this.next = new AtomicReference<Node<E>>(next);
      }
  }
  private AtomicReference<Node<E>> head
      = new AtomicReference<Node<E>>(new Node<E>(null, null));
  private AtomicReference<Node<E>> tail = head;

  public boolean put(E item) {
      Node<E> newNode = new Node<E>(item, null);
      while (true) {
          Node<E> curTail = tail.get();
          Node<E> residue = curTail.next.get();
          if (curTail == tail.get()) {
              if (residue != null)/*A*/
                  tail.compareAndSet(curTail, residue)/*B*/;
              else{
                  if (curTail.next.compareAndSet(null, newNode))/*C*/ {
                      tail.compareAndSet(curTail, newNode)/*D*/ ;
                      return true;
                  }
              }
          }
      }
  }
}
```

3 Wzorce dla współbieżnych i sieciowych obiektów

W książce Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons, 2000 znanej jako POSA2 omówione zostały następujące wzorce projektowe związane ze „współbieżnymi i sieciowymi obiektami".

Obejmują one między innymi:

- Concurrency Patterns
  - Active Object
  - Monitor Object
  - Half-Sync/Half-Async
  - Leader/Followers
  - Thread-Specific Storage



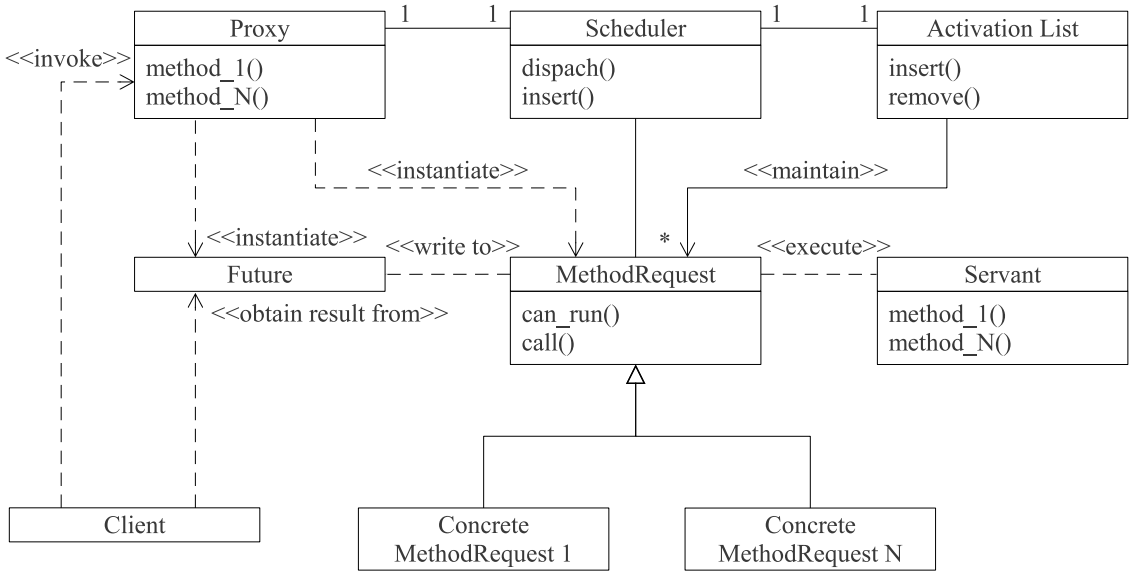
Rysunek 1: Źródło: D. C. Schmidt Pattern-Oriented Software Architectures for Concurrent and Networked Mobile Devices and Clouds D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons, 2000

3.1 Aktywny obiekt (ang. active object)

Cel: Wzorzec aktywny obiekt (ang. active object) rozdziela wykonanie metody od jej wywołania w celu zwiększenia współbieżności i uproszczenia synchronizowanego dostępu do obiektu, który posiada własny wątek.

Rozwiązanie:

1. Dla każdego obiektu, który wymaga współbieżnego wykonania rozdziel wywołanie metody od wykonania metody.
2. Klient ma wrażenie, że wywołuje zwykłą metodę. Natomiast metoda jest automatycznie konwertowana na obiekt żądania metody (ang. method request object). Następnie jest przekazywana do innego wątku gdzie jest konwertowana z powrotem na metodę i wykonywana na obiekcie ją implementującym.
3. Wzorzec aktywny obiekt składa się z następujących elementów:
   - żądanie wywołania metody (ang. method request) jest używane do przekazania kontekstu dotyczącego wywołania danej metody z pośrednika do planisty,
   - pośrednik (ang. proxy) reprezentuje interfejs obiektu, wykonywany jest w wątku klienta, przekształca wywołanie metody w żądanie wywołania metody, zwraca obiekt future,
   - kolejka aktywacji (ang. activation queue) przechowuje w kolejce żądania metod,
   - planista (ang. scheduler) wykonywany w innym wątku niż klienci, zarządza kolejką aktywacji, decyduje które żądanie metody pobrać z kolejki i wykonać przy pomocy sługi implementującego tą metodę,
   - sługa (ang. servant) dostarcza implementacji obiektu, wykonywany jest w wątku planisty,
   - obiekt future umożliwia uzyskanie wyniku przez klienta.



Rysunek 2: Źródło: D. C. Schmidt Pattern-Oriented Software Architectures for Concurrent and Networked Mobile Devices and Clouds D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons, 2000

Obiekty aktywne zostały również opisane w książce Eckel B. Thinking in Java, Wyd. IV, Helion, 2006. Posiadają one następujące cechy:

1. „Każdy obiekt posiada własny wątek wykonania.
2. Każdy obiekt utrzymuje całkowitą kontrolę nad własnymi polami (to wymóg rygorystyczny względem zwykłych klas, które mogą chronić własne pola, ale nie muszą tego robić).
3. Całość komunikacji pomiędzy aktywnymi obiektami odbywa się w formie wymiany komunikatów pomiędzy tymi obiektami.
4. Komunikaty wymieniane pomiędzy obiektami aktywnymi są kolejkowane."

„Cecha aktywności oznacza tu samodzielne zarządzanie własnym wątkiem wykonania i własną kolejką komunikatów oraz założenie kolejkowania wszystkich żądań kierowanych do obiektu tak, aby były obsługiwane po kolei. Obiekty aktywne szeregują komunikaty, a nie metody, co oznacza, że można sobie darować ochronę przed problemami wynikającymi z przerwania wykonania zadania w środku jego pętli. Przesłanie komunikatu do obiektu aktywnego powoduje przekształcenie tego komunikatu na zadanie, które trafia do kolejki obiektu w celu późniejszego uruchomienia."

Przykład pochodzący z Eckel B.: Thinking in Java, Wyd. IV, Helion, 2006

```
//: concurrency/ActiveObjectDemo.java
// Jako argumenty metod asynchronicznych można przekazywać
// stałe, wartości niezmienne, obiekty "rozłączone" albo
// inne obiekty aktywne.
import java.util.concurrent.*;
import java.util.*;

public class ActiveObjectDemo {
    private ExecutorService ex = Executors.newSingleThreadExecutor();
    private Random rand = new Random(47);

    // Losowe opóźnienie, dające efekt
    // prowadzenia długotrwałych obliczeń:
    private void pause(int factor) {
        try {
            TimeUnit.MILLISECONDS.sleep(100 + rand.nextInt(factor));
```

```
        } catch (InterruptedException e) {
            System.out.print("Przerwanie zadania w sleep()");
        }
    }

    public Future<Integer> calculateInt(final int x, final int y) {
        return ex.submit(new Callable<Integer>() {
            public Integer call() {
                System.out.print("zaczynam " + x + " + " + y);
                pause(500);
                return x + y;
            }
        });
    }

    public Future<Float> calculateFloat(final float x, final float y) {
        return ex.submit(new Callable<Float>() {
            public Float call() {
                System.out.print("zaczynam " + x + " + " + y);
                pause(2000);
                return x + y;
            }
        });
    }

    public void shutdown() {
        ex.shutdown();
    }

    public static void main(String[] args) {
        ActiveObjectDemo d1 = new ActiveObjectDemo();
        // Blokada wyjątku ConcurrentModificationException:
        List<Future<?>> results = new CopyOnWriteArrayList<Future<?>>();
        for (float f = 0.0f; f < 1.0f; f += 0.2f)
            results.add(d1.calculateFloat(f, f));
        for (int i = 0; i < 5; i++)
            results.add(d1.calculateInt(i, i));
        System.out.print("Wykonano wszystkie wywołania asynchroniczne");
        while (results.size() > 0) {
            for (Future<?> f : results)
                if (f.isDone()) {
                    try {
                        System.out.print(f.get());
                    } catch (Exception e) {
                        throw new RuntimeException(e);
                    }
                    results.remove(f);
                }
        }
        d1.shutdown();
    }
}
```

„Aby dyskretnie zapobiec nadmiernym powiązaniom pomiędzy obiektami, wszelkie argumenty przekazywane do metod obiektu aktywnego powinny być wartościami niemodyfikowalnymi albo innymi obiektami aktywnymi, ewentualnie obiektami rozłączonymi (to mój własny termin), czyli takimi, które nie mają powiązania z żadnym innym zadaniem (trudno zapewnić ten wymóg, a to z powodu braku wsparcia ze strony języka) [4].”

Przykład użycia dynamicznego proxy pozwalającego poprzez metodę invoke() na zmianę sposobu wykonania metody:

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyFactory {
    static class MyInvocationHandler<T> implements InvocationHandler {
        private T object;

        MyInvocationHandler(T object) {
            this.object = object;
        }

        @Override public String toString(){
            return "" + this.getClass();
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
            System.out.print("Proxy->");
            return method.invoke(object, args);
        }
    }

    public static <T> T createProxy(Class<T> clazz, T object) {
        MyInvocationHandler<T> proxy = new MyInvocationHandler<T>(object);
        return (T) Proxy.newProxyInstance(Executable.class.getClassLoader(),
                new Class[] { clazz }, (InvocationHandler) proxy);
    }
}
```

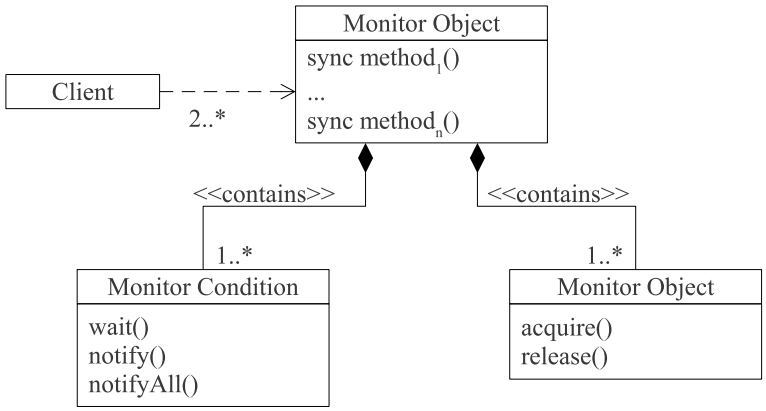
```
interface Executable {
    public void method();
}

class MyExecutable implements Executable {
    @Override
    public void method() {
        System.out.println("Method");
    }
}

public class TestProxy {
    public static void main(String[] args) {
        Executable object = new MyExecutable();
        Executable proxy = ProxyFactory.createProxy(Executable.class, object);
        object.method();
        proxy.method();
    }
}
```

3.2 Obiekt monitora (ang. monitor object)

Cel: Wzorzec obiekt monitora (ang. monitor object) gwarantuję, że w danym momencie możliwe jest wywołanie tylko jednej metody danego obiektu. Umożliwia również metodom obiektu wspólnie ustalanie kolejność ich wywołania.

Rozwiązanie:

- Każdy obiektu wykorzystywany współbieżnie przez wątki klienckie zdefiniuj jako obiekt monitora.
- Dostęp do obiektu monitora powinien być możliwy tylko poprzez metody sychronizowane.
- W danym momencie możliwe jest wywołanie tylko jednej metody obiektu monitora.
- Każdy obiekt monitora powinien zawierać blokadę monitora (ang. monitor lock), która jest używana przez metody synchronizowane do serializacji dostępów do obiektu.
- Wykorzystując warunki monitora (ang. monitor conditons) skojarzone z obiektem monitora metody synchronizowane mogą określić w jakich warunkach zawieszają oraz wznawiają one swoje działanie.



Rysunek 3: Źródło: D. C. Schmidt Pattern-Oriented Software Architectures for Concurrent and Networked Mobile Devices and Clouds D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons, 2000

W Javie wariant wzorca monitora można znaleźć w klasie Object:

- public final void wait() – wątek zostaje zatrzymany, aż do wywołania metody notify() lub notifyAll(), istnieją także przesłonięte wersje tej metody ograniczające maksymalny czas czekania (public final void wait(long timeout)oraz public final void wait(long timeout, int nanos)),
- public final void notify() oraz public final void notifyAll(), budzi wątek (lub wątki) czekające na monitorze obiektu.

Metody wait() oraz notify() mogą być wywoływane tylko wewnątrz bloku lub metody synchronizowanej.

```
class Buffer {
    int value;
}
```

```
public class Test extends Thread {

    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        WriterThread writer = new WriterThread(buffer);
        ReaderThread reader = new ReaderThread(buffer);
        reader.start();
        writer.start();
    }
}
```

```
class WriterThread extends Thread {
    final private Buffer b;
    private int i = 0;

    WriterThread(Buffer b) {
        this.b = b;
    }

    public void run() {
        try {
            while (true) {
                synchronized (b) {
                    int t = ++i;
                    b.value = t;
                    Thread.sleep(t);
                    System.err.println("< WriterThread " + b.value);
                    b.notify();
                    System.err.println("<<");
                    b.wait();
                    System.err.println("<<<");
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

```
class ReaderThread extends Thread {
    final private Buffer b;

    ReaderThread(Buffer b) {
        this.b = b;
    }

    public void run() {
        try {
            while (true) {
                synchronized (b) {
                    System.err.println(">");
                    b.wait();
                    System.err.println(">> ReaderThread " + b.value);
                    b.notify();
                    System.err.println(">>>");
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

W Javie wariant wzorca monitora można z znaleźć w interfejsie java.util.concurrent.locks.Condition, który zawiera metody:

- await(), await(long time, TimeUnit unit), awaitNanos(long nanosTimeout), awaitUninterruptibly(), awaitUntil(Date deadline),

- signal(),
- signalAll().

3.3 Half-Sync / Half-Async

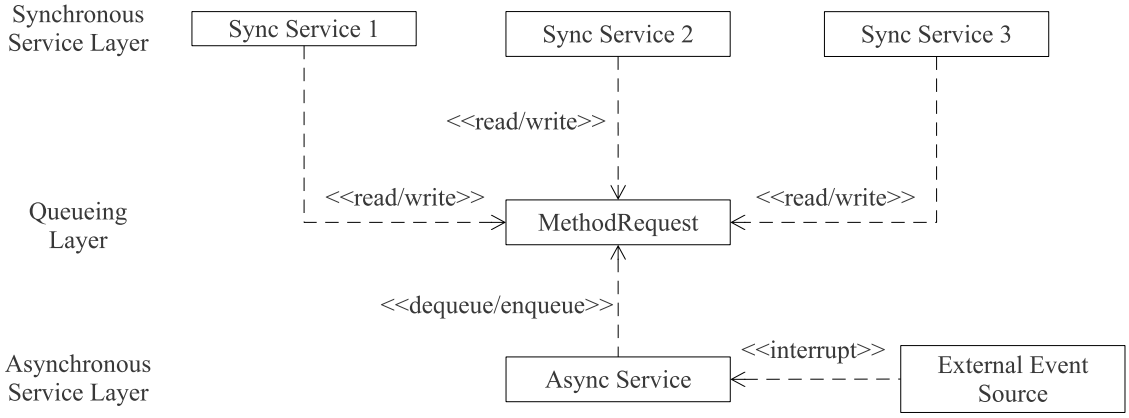
Cel: Wzorzec Half-Sync / Half-Async ma za zadanie rozdzielić synchroniczne wejście / wyjście od asynchronicznego wejścia / wyjścia w systemie w celu uproszczenia programowania współbieżnego, ale w taki sposób, aby nie wprowadzać spadku wydajności.

Rozwiązanie:

- Zadania wysokopoziomowe używają synchronicznego modelu wejścia / wyjścia, który upraszcza programowanie współbieżne.
- Zadania niskopoziomowe używają asynchronicznego modelu wejścia / wyjścia, który zwiększa wydajność wykonywania.
- Komunikacja pomiędzy synchroniczną i asynchroniczną warstwą realizowana jest przez warstwę kolejkowania.

Zastosowanie:

1. System posiada następujące cechy:
   - musi przetwarzać zadania w odpowiedzi na zewnętrzne zdarzenia, które pojawiają się asynchronicznie oraz
   - byłoby nieefektywnym przydzielenie oddzielnego wątku do wykonywania synchronicznych operacji wejścia / wyjścia dla każdego źródła zewnętrznych zdarzeń oraz
   - wysokopoziomowe zadania w systemie mogą zostać znacznie uproszczone jeżeli operacje wejścia / wyjścia będą wykonywanie synchronicznie.
2. Jedno lub więcej zadań w systemie musi być wykonywane w pojedynczym wątku (ang. single thread of control), natomiast inne mogą zyskać jeżeli będą wykonywane wielowątkowo.



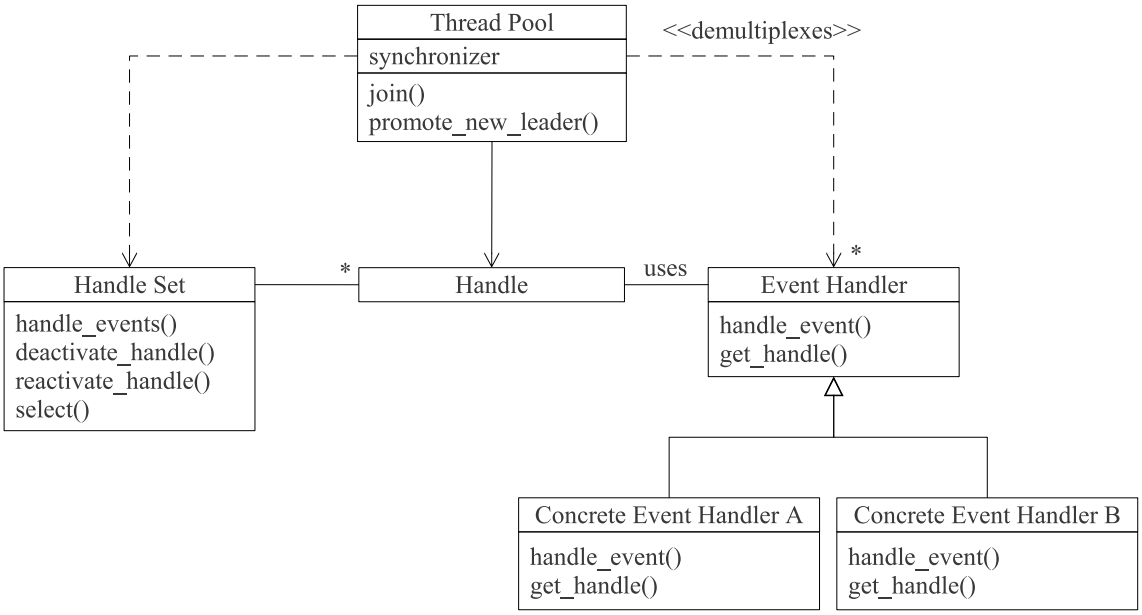
Rysunek 4: Źródło: D. C. Schmidt Pattern-Oriented Software Architectures for Concurrent and Networked Mobile Devices and Clouds D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons, 2000

3.4 Leader/Followers

Cel: Wzorzec architektoniczny Leader/Followers dostarcza wydajny model współbieżności, w którym wiele wątków zmienia się na przemian (ang. take turns) współdzieląc zbiór źródeł zdarzeń w celu wykrywania, demultipeksowania, rozdzielanie oraz przetwarzania żądań usług, które występują dla tych źródeł zdarzeń.

Kontekst: Aplikacja sterowana zdarzeniami, w której wiele żądań usługi zawartych w zdarzeniach przybywa ze zbioru źródeł danych i musi być przetwarzanych wydajnie przez wiele wątków, które współdzielą te źródła danych.

Rozwiązanie: Stwórz strukturę puli wątków w celu wydajnego współdzielenia zbioru źródeł zdarzeń poprzez zmienianie się na przemian (ang. taking turns) demultipeksując zdarzenia pochodzących z tych źródeł zdarzeń i synchronicznie przydzielając zdarzenia do usług aplikacji, które je przetwarzają.



Rysunek 5: Źródło: D. C. Schmidt Pattern-Oriented Software Architectures for Concurrent and Networked Mobile Devices and Clouds D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons, 2000

Informacje na temat wzorca Leader-Follower można również znaleźć w książce: Java Threads, 3rd Edition, Understanding and Mastering Concurrent Programming – Scott Oaks, Henry Wong, O'Reilly, 2009. Implementacja tego wzorce w Javie może zostać uproszczone ponieważ wewnętrzna implementacja metody accept() z klasy ServerSocket jest synchronizowana [1]. Powoduje to, że w danym momencie tylko jeden wątek może wywołać tę metodę, a pozostałe zostaną zablokowane.

Poniższy przykład pochodzi z tej książki:

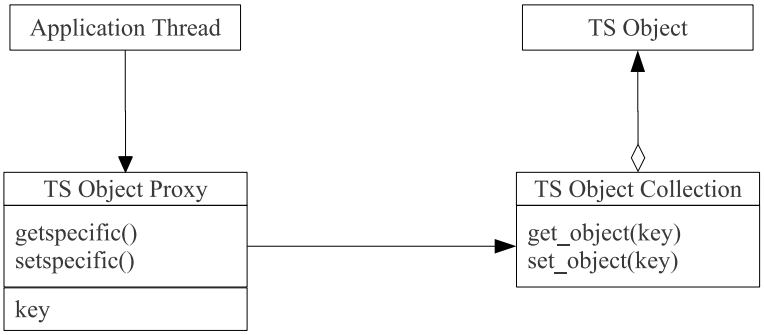
```
package javathreads.examples.ch12;

import java.net.*;
import java.io.*;

public abstract class TCPThrottledServer implements Runnable {
    ServerSocket server = null;
    Thread[] serverThreads;
    volatile boolean done = false;

    public synchronized void startServer(int port, int nThreads) throws IOException {
        server = new ServerSocket(port);

        serverThreads = new Thread[nThreads];
        for (int i = 0; i < nThreads; i++) {
            serverThreads[i] = new Thread(this);
            serverThreads[i].start();
```

[1] Klasa: java.net.PlainSocketImpl metoda: protected synchronized void accept(SocketImpl s) throws IOException

```
        }
    }

    public synchronized void setDone() {
        done = true;
    }

    public void run() {
        while (!done) {
            try {
                Socket data;
                data = server.accept();
                run(data);
            } catch (IOException ioe) {
                System.out.println("Accept error " + ioe);
            }
        }
    }

    public void run(Socket data) {
    }
}
```

3.5 Thread-Specific Storage

Cel: Zapewnienie wielu wątkom globalnie dostępnego punktu dostępu do pobierania prywatnych danych bez konieczności ponoszenia narzutu związanego z blokowaniem przy każdym dostępie.



Rysunek 6: Źródło: D. C. Schmidt Pattern-Oriented Software Architectures for Concurrent and Networked Mobile Devices and Clouds D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons, 2000

Koncepcja zbliżona do Thread-Specific Storage została zaimplementowana w Javie w klasie java.lang.ThreadLocal<T>.

```
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
```

```
        return setInitialValue();
    }

    public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }
```

Kod pochodzi z pliku źródłowego java.lang.ThreadLocal.java dołączonego do Java Development Kit.

4 Frameworki aplikacji sieciowych

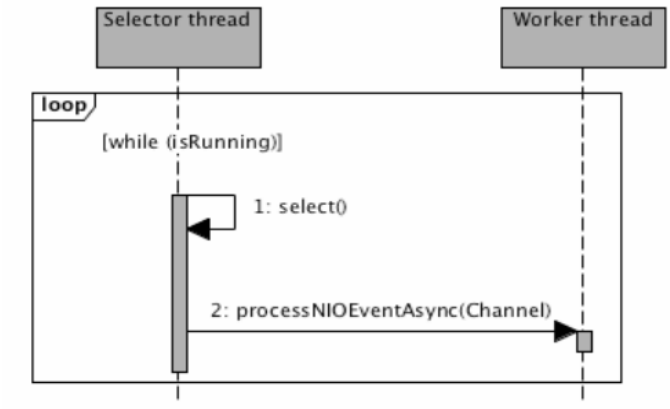
Przykłady framework'ów aplikacji sieciowych (https://mina.apache.org/mina-project/related-projects.html):

- Apache MINA (https://mina.apache.org/),
- Project Grizzly (https://javaee.github.io/grizzly/),
- Netty (https://netty.io/),
- NIO Framework (https://sourceforge.net/projects/nioframework/),
- QuickServer (http://www.quickserver.org/).

We frameworku Grizzly przetwarzanie zdarzeń NIO, które pojawiło się na kanale NIO może być zgodne z jedną z poniższych strategii [https://javaee.github.io/grizzly/iostrategies.html]:

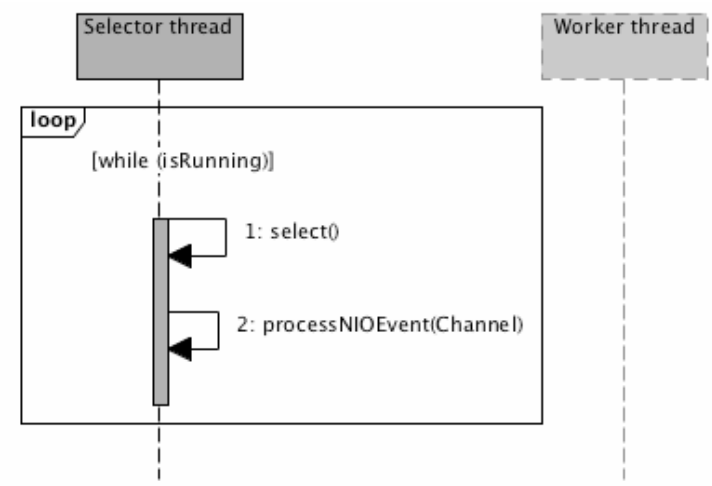
- Worker-thread IOStrategy,
- Same-thread IOStrategy,
- Dynamic IOStrategy,
- Leader-follower IOStrategy.

Strategia Worker-thread IOStrategy polega na delegowaniu przez wątek selektora przetwarzania zdarzeń NIO do wątków roboczych. Istnieje możliwość zmiany rozmiaru puli wątków selektora i roboczych.



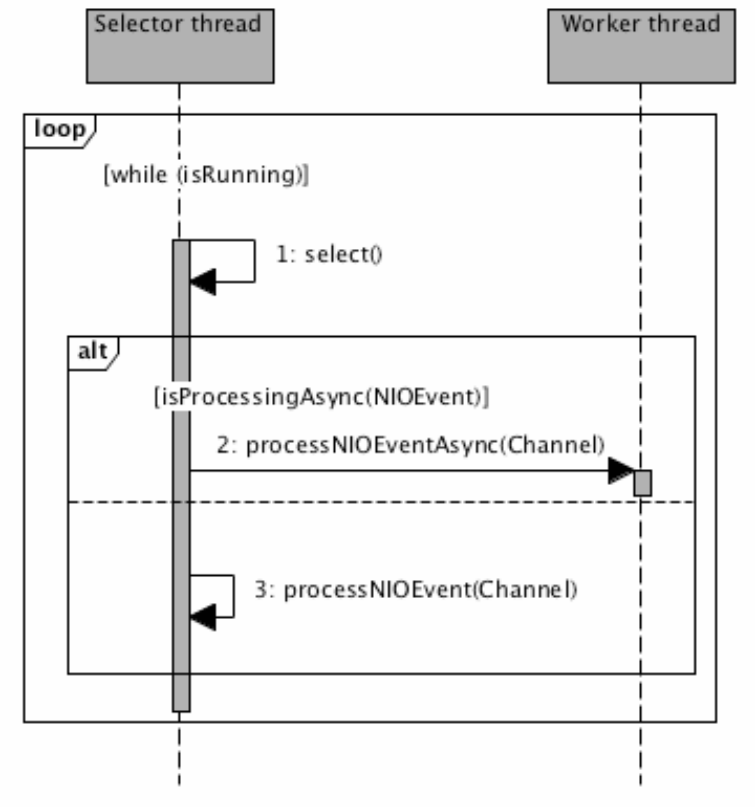
Rysunek 7: Źródło: https://javaee.github.io/grizzly/iostrategies.html

Strategia Same-thread IOStrategy polega na przetwarzaniu zdarzeń NIO przez ten sam wątek, który operował na selektorze. Istnieje możliwość zmiany rozmiaru puli wątków selektora.



Rysunek 8: Źródło: https://javaee.github.io/grizzly/iostrategies.html

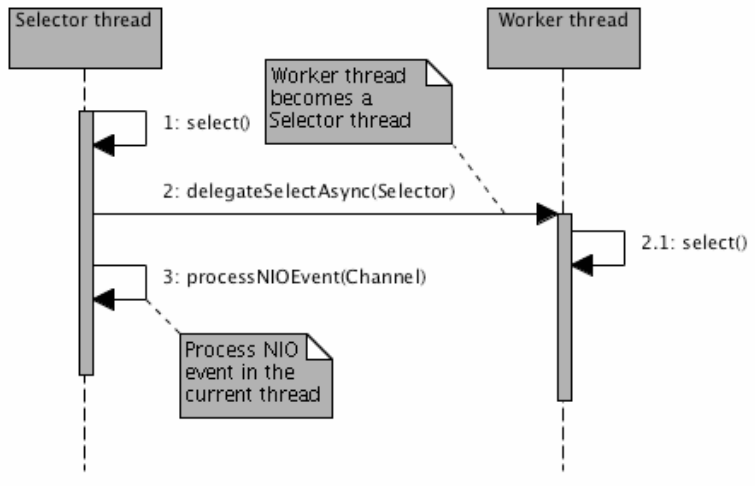
Strategia Dynamic IOStrategy polega na przełączaniu się w czasie wykonania pomiędzy Same-thread IOStrategy oraz Worker-thread IOStrategy zależnie od aktualnych warunków.



Rysunek 9: Źródło: https://javaee.github.io/grizzly/iostrategies.html

Strategia Leader-follower IOStrategy polega na zamianie wątku roboczego w wątek selektora poprzez przekazaniu mu kontroli nad selektorem. Powoduje to, że przetwarzanie zdarzenia NIO jest wykonywane

w aktualnym wątku.



Rysunek 10: Źródło: https://javaee.github.io/grizzly/iostrategies.html

5 Przykłady pokazujące różne warianty implementacji serwera echa

Listing 1: Klasa opisująca zadanie do przetwarzania po stronie serwera echa {src/EchoTask.java}

```
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.concurrent.TimeUnit;

public class EchoTask implements Runnable {

    private Socket socket;

    public EchoTask(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            InputStream is = socket.getInputStream();
            byte b = (byte) is.read();

            TimeUnit.SECONDS.sleep(1);

            OutputStream os = socket.getOutputStream();
            os.write(b);
            os.flush();
            System.out.println("Read = " + b + " Write = " + b);

            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

Listing 2: Abstrakcyjna klasa serwera echa {src/EchoServer.java}

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.util.concurrent.ExecutorService;

public abstract class EchoServer implements Runnable {

    private ServerSocket server;
    private ExecutorService executor;

    public EchoServer(ExecutorService executor) {
        if (executor == null)
            throw new NullPointerException();
        this.executor = executor;
        try {
            server = new ServerSocket(54321);
            server.setSoTimeout(10_000);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        System.out.println("Server startup");

        try {
            for (;;) {
                Socket socket = server.accept();
                process(socket);
            }

        } catch (SocketTimeoutException e) {
            System.out.println("Server shutdown");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            shutdown();
        }
    }

    protected void process(Socket socket) {
        EchoTask task = new EchoTask(socket);
        executor.execute(task);
    }

    protected void shutdown() {
        executor.shutdown();
    }
}
```

Listing 3: Uproszczona wersja interfejsu ExecutorService {src/SimpleExecutorService.java}

```
import java.util.Collection;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
```

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public interface SimpleExecutorService extends ExecutorService {

    public void execute(Runnable task);

    public void shutdown();

    public default List<Runnable> shutdownNow() {
        throw new UnsupportedOperationException();
    }

    public default boolean isShutdown() {
        throw new UnsupportedOperationException();
    }

    public default boolean isTerminated() {
        throw new UnsupportedOperationException();
    }

    public default boolean awaitTermination(long timeout, TimeUnit unit) throws ↩
         InterruptedException {
        throw new UnsupportedOperationException();
    }

    public default <T> Future<T> submit(Callable<T> task) {
        throw new UnsupportedOperationException();
    }

    public default <T> Future<T> submit(Runnable task, T result) {
        throw new UnsupportedOperationException();
    }

    public default Future<?> submit(Runnable task) {
        throw new UnsupportedOperationException();
    }

    public default <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws ↩
         InterruptedException {
        throw new UnsupportedOperationException();
    }

    public default <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long ↩
         timeout, TimeUnit unit)
            throws InterruptedException {
        throw new UnsupportedOperationException();
    }

    public default <T> T invokeAny(Collection<? extends Callable<T>> tasks)
            throws InterruptedException, ExecutionException {
        throw new UnsupportedOperationException();
    }

    public default <T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit↩
          unit)
            throws InterruptedException, ExecutionException, TimeoutException {
        throw new UnsupportedOperationException();
    }
}
```

Listing 4: Konkretna klasa serwera echa z zadaniami uruchamianymi w tym samym wątku {src/TheSameThreadEchoServer.java}

```
public class TheSameThreadEchoServer extends EchoServer {

    public TheSameThreadEchoServer(TheSameThreadExecutor executor) {
        super(executor);
    }

    public static void main(String args[]) {
        TheSameThreadEchoServer server = new TheSameThreadEchoServer(new TheSameThreadExecutor());
        server.run();
    }

}

class TheSameThreadExecutor implements SimpleExecutorService {

    @Override
    public void execute(Runnable task) {
        task.run();
    }

    @Override
    public void shutdown() {
    }

}
```

Listing 5: Konkretna klasa serwera echa z zadaniami uruchamianymi w innych wątkach {src/OtherThreadEchoServer.java}

```
import java.util.HashSet;
import java.util.Set;

public class OtherThreadEchoServer extends EchoServer {

    public OtherThreadEchoServer(OtherThreadExecutor executor) {
        super(executor);
    }

    public static void main(String args[]) {
        OtherThreadEchoServer server = new OtherThreadEchoServer(new OtherThreadExecutor());
        server.run();
    }

}

class OtherThreadExecutor implements SimpleExecutorService {

    private Set<Thread> threads = new HashSet<>();

    @Override
    public void execute(Runnable task) {
        Thread newThread = new Thread(task);
        threads.add(newThread);
        newThread.start();
    }

    @Override
    public void shutdown() {
        for(Thread t : threads) {
```

```
            if(t.isAlive())
                t.interrupt();
        }
    }

}
```

Listing 6: Konkretna klasa serwera echa z jednowątkową pulą {src/SingleThreadEchoServer.java}

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SingleThreadEchoServer extends EchoServer {

    public SingleThreadEchoServer(ExecutorService executor) {
        super(executor);
    }

    public static void main(String args[]) {
        SingleThreadEchoServer server = new SingleThreadEchoServer(Executors.↩
            newSingleThreadExecutor());
        server.run();
    }

}
```

Listing 7: Konkretna klasa serwera echa z cache'owaną pulą wątków {src/CachedThreadEchoServer.java}

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CachedThreadEchoServer extends EchoServer {

    public CachedThreadEchoServer(ExecutorService executor) {
        super(executor);
    }

    public static void main(String args[]) {
        CachedThreadEchoServer server = new CachedThreadEchoServer(Executors.newCachedThreadPool()↩
            );
        server.run();
    }

}
```

Listing 8: Konkretna klasa serwera echa z pulą wątków o określonym rozmiarze {src/FixedThreadEchoServer.java}

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class FixedThreadEchoServer extends EchoServer {

    public FixedThreadEchoServer(ExecutorService executor) {
        super(executor);
    }

    private static final int NUMBER_OF_THEADS = 3;

    public static void main(String[] args) {
```

```
        FixedThreadEchoServer server = new FixedThreadEchoServer(Executors.newFixedThreadPool(↩
            NUMBER_OF_THEADS));
        server.run();
    }

}
```

Listing 9: Klasa klienta echa {src/EchoClient.java}

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.time.Duration;
import java.time.Instant;
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class EchoClient {

    private static final int NUMBER_OF_CLIENTS = 16;

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newCachedThreadPool();
        Instant start = Instant.now();
        for (int i = 0; i < NUMBER_OF_CLIENTS; i++) {
            executor.execute(new Client());
        }
        executor.shutdown();
        executor.awaitTermination(1, TimeUnit.DAYS);
        Instant finish = Instant.now();
        long timeMiliis = Duration.between(start, finish).toMillis();
        System.out.println("Time (ms): " + timeMiliis);

    }
}

class Client implements Runnable {
    private Random r = new Random();

    @Override
    public void run() {
        try (Socket socket = new Socket("localhost", 54321);
                OutputStream os = socket.getOutputStream();
                InputStream is = socket.getInputStream();) {

            byte bWrite = (byte) r.nextInt(Byte.MAX_VALUE);

            os.write(bWrite);

            byte bRead = (byte) is.read();
            System.out.println("Write = " + bWrite + " Read = " + bRead);

        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }
}
```

Literatura

[1] Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D., Java Współbieżność dla praktyków, Helion 2007

[2] Horstmann C.S., Cornell G., Java Podstawy, Helion, Wyd. VIII, 2009

[3] Horstmann C.S., Cornell G., Java Techniki zaawansowane, Helion, Wyd. VIII, 2009

[4] Eckel B.: Thinking in Java, Wyd. IV, Helion, 2006.

[5] Bloch J.: Java Efektywne programowanie, Wyd. II, Helion, 2009.

[6] Oaks S., Wong H.: Java Threads, 3rd Edition, Understanding and Mastering Concurrent Programming, O'Reilly, 2009

[7] Silberschatz A., Galvin P. B., Gagne G.: Podstawy systemów operacyjnych, WNT, 2005

[8] The Java Language Specification, Third Edition, http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

[9] http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html