Uniwersytet Jagielloński w Krakowie
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Piotr Stokłosa

Nr albumu: 1207293

System do monitorowania wątków w języku Java

Praca magisterska
na kierunku Informatyka Stosowana

Praca wykonana pod kierunkiem
dra Piotra Oramusa
Instytut Informatyki Stosowanej
Zakład Technologii Informatycznych

Kraków 2025

Streszczenie

System do monitorowania wątków w języku Java (Thread Agent) umożliwia śledzenie działania wątków aplikacji, dostarczając kluczowych informacji dotyczących obiektów synchronizacyjnych oraz przebiegu pracy poszczególnych wątków. Na podstawie wygenerowanych plików logowania użytkownik może zidentyfikować przyczyny problemów związanych z synchronizacją, wykryć potencjalne wycieki wątków oraz przeanalizować wzajemne przeplatanie się wątków. Thread Agent posiada mechanizm analizujący i wykrywający zakleszczenia wątków, wyjątki oraz inne kryzysowe sytuacje istotne dla użytkownika. Aplikacja ma na celu nie tylko wspomaganie rozwiązywania problemów w środowiskach wielowątkowych, ale również pełni funkcję edukacyjną, pomagając użytkownikowi lepiej zrozumieć mechanizmy działania współbieżności w Javie.

Thread monitoring system in Java (Thread Agent) enables tracking the behavior of application threads, providing key insights into synchronization objects and the execution flow of individual threads. Based on generated log files, the user can identify the causes of synchronization issues, detect potential thread leaks, and analyze thread interleaving. Thread Agent features a mechanism that analyzes and detects thread deadlocks, exceptions, and other critical situations relevant to the user. The application is designed not only to support debugging in multithreaded environments but also to serve an educational purpose, helping users better understand the mechanisms of concurrency in Java.

Spis treści

Plan pracy

- Pierwszy rozdział zawiera wstęp do pracy. Przedstawia problem, cel pracy oraz analizuje dostępne narzędzia, które służą, podobnie jest stworzona w ramach pracy magisterskiej aplikacja Thread Agent, pomocą przy aplikacjach wielowątkowych w Javie.
- Drugi rozdział poświęcony jest wątkom w Javie. Omawia ich zalety i typowe problemy związane z programowaniem wielowątkowym oraz zagadnienia synchronizacji pracy wątków. Przedstawia również narzędzia udostępniane przez język Java, które ułatwiają tworzenie aplikacji wielowątkowych.
- Trzeci rozdział opisuje mechanizm Java agentów oraz interfejs `Instrumentation`. Zawiera również informacje na temat popularnych bibliotek w kontekście agentów Java oraz porusza kwestie związane z ładowaniem klas.
- W czwartym rozdziale znajduje się opis techniczny i praktyczny Thread Agenta. Przedstawia sposób działania aplikacji oraz instrukcję jej użytkowania.
- Piąty rozdział prezentuje przykładowe aplikacje uruchomione z użyciem Thread Agenta, ilustrując praktyczne zastosowanie projektu w różnych scenariuszach.
- Szósty rozdział przedstawia propozycje dalszego rozwoju aplikacji oraz pomysły na usprawnienia i rozszerzenia funkcjonalności Thread Agenta.
- W siódmym, ostatnim rozdziale znajduje się podsumowanie całej pracy.

Rozdział 1

Wstęp

1.1 Przedstawienie problemu

Współczesne aplikacje często wykorzystują wielowątkowość w celu zwiększenia wydajności oraz uporządkowania zadań w projekcie. Mimo często lepszego wykorzystania zasobów sprzętowych, programowanie wielowątkowe wiąże się z wieloma trudnościami. Problemy, które wynikają z błędów implementacyjnych, są często trudne do wykrycia, zrozumienia i znalezienia, szczególnie w złożonych systemach. Dostępne narzędzia dostarczają jedynie częściowych informacji o pracy wątkach. Często wskazują jedynie, że aplikacja nie działa prawidłowo, nie dostarczając przy tym wystarczających informacji umożliwiających zlokalizowanie źródła problemu. Brakuje rozwiązań, które nie tylko pokazują aktualny stan problemu w aplikacji, lecz także są w stanie śledzić i na bieżąco rejestrować wykonywane przez wątki operacje.

1.2 Cel pracy

Celem niniejszej pracy magisterskiej jest wykonanie aplikacji wspomagającej zrozumienie działania mechanizmów wielowątkowych w Javie oraz pomagającej wykryć problemy z nimi związane. Program koncentruje się na rejestrowaniu działania poszczególnych wątków, które mogą pomóc w identyfikacji problemu. Dodatkowo Thread Agent monitoruje klasy odpowiedzialne za zarządzanie wątkami, a także w przypadku zamknięcia przedwcześnie programu, zapisuje informację o niezakończonych wątkach. Na podstawie analizy istniejących rozwiązań zaproponowano własne podejście do problemu z innej perspektywy, wykorzystując Java Agenta. Takie podejście gwarantuje uzyskanie dodatkowych informacji, które mogą być kluczowe w monitorowaniu aplikacji. Projekt ma również wymiar edukacyjny – umożliwia lepsze zrozumienie mechanizmów współbieżności poprzez obserwację ich działania z perspektywy programu.

1.3 Przegląd dostępnych rozwiązań

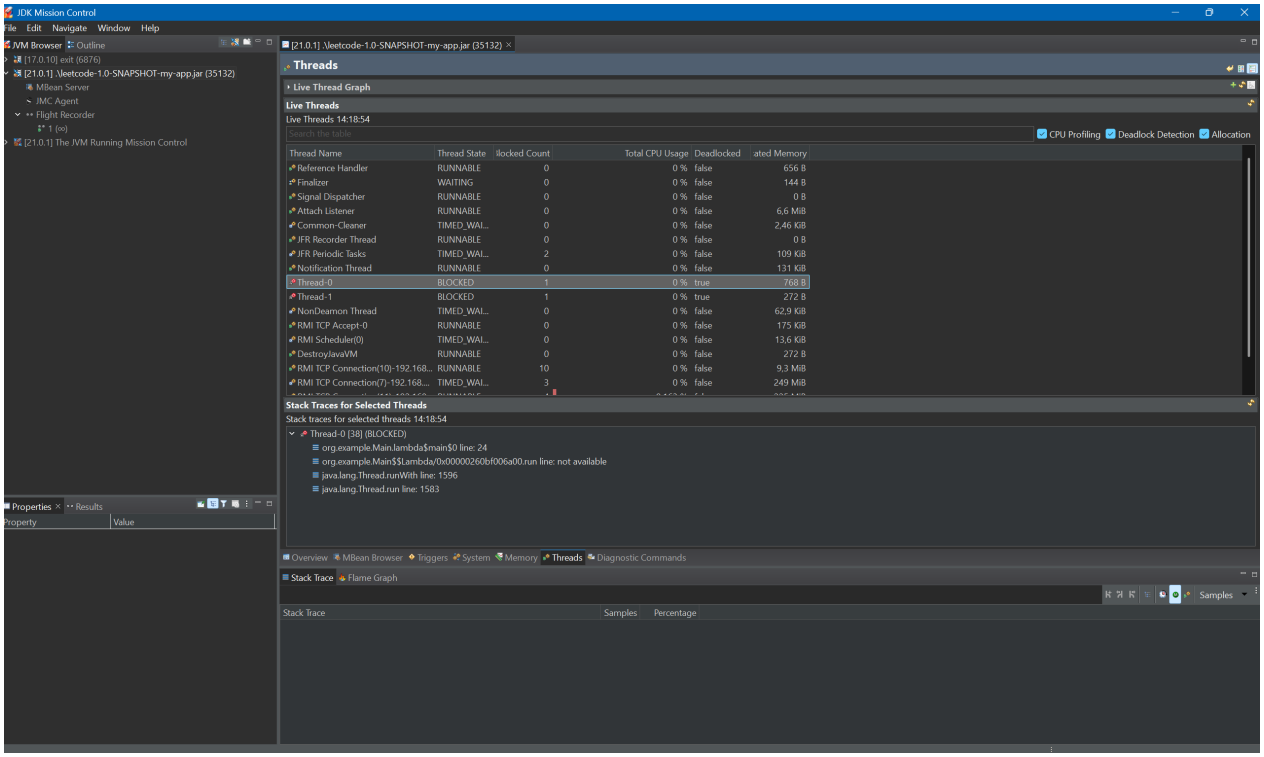
1.3.1 Zrzut stanu wątków

Najpopularniejszą metodą wykrywania błędu jest ręczne sprawdzenie wątków w zrzucie stanu wątków. Można go uzyskać na kilka sposobów. W przypadku posiadania pełnego środowiska Java (JDK), można użyć narzędzia `jstack <PID>` podając identyfikator procesu JVM. W przypadku posiadania wyłącznie środowiska uruchomieniowego Java (JRE), można w systemie Linux wysłać do procesu JVM sygnał `SIGQUIT` używając instrukcji `kill -3 <PID>`. W zrzucie wątków znajduje się aktualny stos wywołań oraz stan każdego z wątków. Za pomocą takich informacji można wykryć zakleszczenie (sytuacja, w której wątki czekają na siebie nawzajem), natomiast często jest to niewystarczające. Dodatkowo wątki stworzone przez obiekty implementujące interfejs `ExecutorService` posiadają własne ścieżki wywołań, co powoduje, że miejsce utworzenia wątków może być nieznane i niewidoczne w zrzucie wątków.

Zrzut stanu wątków jest najprostszym narzędziem do analizy, gdy wystąpi problem ze współbieżnością, ale jest on ograniczony o brak możliwości zobaczenia historii przebiegu pracy wątku. Wątki tworzone przez `ExecutorService` posiadają bardzo mało informacji. Dodatkowo wykonanie zrzutu stanu wątków może być kosztowne pod względem wydajności i zamrozić aplikację na pewien okres czasu.

1.3.2 JDK Mission Control

Program JDK Mission Control (JMC)[1] od firmy Oracle[2], współpracując z Java Flight Recorder[3], zbiera szczegółowe informacje o działaniu JVM. Rysunek 1.1 przedstawia widok wątków przykładowej aplikacji w JMC, w której doszło do zakleszczenia. JMC pozwala na przeglądanie między innymi stanu wątków, wykrywanie zakleszczeń oraz analizę incydentów po ich wystąpieniu. Pozwala zobaczyć, w jakim stopniu każdy wątek obciąża procesor. Wadą tego rozwiązania jest mało dostępnych informacji na temat wątków, brak możliwości sprawdzenia, gdzie dany wątek został stworzony, brak historii wywołania wielu ważnych metod, które mogły doprowadzić do sytuacji prezentowanej w graficznym interfejsie oraz brak monitorowania obiektów `ExecutorService`.



Rysunek 1.1: Widok wątków w JDK Mission Control

Rozdział 2

Wielowątkowość

2.1 Wstęp do wątków

Podczas pisania aplikacji programiści napotykają problemy związane z zaprojektowaniem systemu, aby działał logicznie, spójnie oraz szybko. Często decydują się na użycie wątków. Wątki służą do wykonywania wielu zadań jednocześnie. Każdy program jest uruchamiany z co najmniej jednym wątkiem. Użycie wielu wątków w aplikacji tworzy program współbieżny, czyli taki, który posiada zdolność zarządzania wieloma zadaniami jednocześnie, niezależnie od tego, czy są one wykonywane na maszynie w tym samym czasie. Wątki mogą współdzielić zasoby. Zadaniem programisty jest tak zsynchronizować dostęp do zasobów, aby uniknąć występowania konfliktów i błędów w działaniu aplikacji.

2.2 Zalety stosowania wątków

Poniżej przedstawiono główne zalety wynikające z zastosowania wielowątkowości w aplikacjach:

- **Wydajność**

  Na komputerach wielordzeniowych, programy współbieżne mogą wykonywać się równolegle, co przyspiesza działanie aplikacji czasami nawet kilkunastokrotnie.

- **Responsywność**

  Aplikacje wielowątkowe nie "zamrażają się" tak często ze względu na oddelegowanie czasochłonnych zadań, jak na przykład pobieranie danych, do innych wątków, które działają w tle.

- **Podział zadań**

  Programista może podzielić zadania między wątki, przez co każdy wątek będzie odpowiedzialny za inne zadanie.

- **Cykliczność**

  Za pomocą wątków, programista może zaprogramować zadanie, które jest wykonywane cyklicznie.

- **Asynchroniczność**

  Brak konieczności blokowania głównego wątku aplikacji do wykonywania zadań.

2.3 Wady stosowania wątków

Poniżej przedstawiono główne wady i ograniczenia związane z wykorzystaniem wielowątkowości w aplikacjach:

- **Złożoność kodu**

  Wielowątkowy kod jest znacznie trudniejszy do napisania, testowania oraz utrzymywania. Zrozumienie działania klas i metod odpowiedzialnych za tworzenie i zarządzanie wątkami jest znacznie trudniejsze, niż napisanie jednowątkowej (sekwencyjnej) aplikacji.

- **Przełączanie kontekstu**

  Gdy liczba wątków przewyższa liczbę dostępnych sprzętowo wątków procesora, system operacyjny wraz z wirtualną maszyną Javy będzie przełączał kontekst, co powoduje dodatkowe koszty, a tym samym spadek wydajności.

- **Zużycie pamięci**

  Każdy wątek zużywa dodatkową pamięć.

2.4 Synchronizacja

Synchronizacja wątków to mechanizm, który umożliwia bezpieczne współdzielenie danych między wątkami. Głównym zadaniem programisty jest zapewnienie spójności danych, co oznacza, że dane współdzielone przez wiele wątków pozostają w poprawnym i przewidywalnym stanie. Jest to wyzwanie dla programisty, który podchodzi do napisania kodu wielowątkowego. Każdy błąd może być czasochłonny do wykrycia, ze względu na trudne do przetestowania przypadki czy scenariusze występujące bardzo rzadko.

Główne problemy wynikające z braku lub błędnej synchronizacji[4]:

- **Problem widoczności**

  Zmiany, które dokona jeden wątek mogą być niewidoczne dla wątku drugiego, z powodu zaktualizowania lokalnego cache'a, a nie pamięci RAM.

- **Zakleszczenie (ang. deadlock)[5]**

  Sytuacja, w której dwa lub więcej wątków czeka na zasoby trzymane przez siebie nawzajem, przez co żaden z nich nie może kontynuować działania. Każdy z wątków blokuje zasób, którego potrzebuje inny wątek, tworząc cykliczną zależność i uniemożliwiając dalsze postępy programu bez interwencji z zewnątrz.

- **Livelock[6]**

  Wątki cały czas zmieniają swój stan w odpowiedzi na siebie nawzajem, przez co nie wykonują użytecznej pracy.

- **Zagłodzenie (ang. starvation)[7]**

  Wątek nigdy nie dostaje dostępu do zasobu, bo inne wątki ciągle go przetrzymują.

- **Warunek wyścigu (ang. race condition)[8]**

  Dwa lub więcej wątków uzyskują dostęp do wspólnych danych w tym samym czasie. Wynik nawet prostej operacji staje się nieprzewidywalny.

2.5 Wątki w Javie

Wątek w Javie jest reprezentowany jako obiekt klasy `Thread`. Oznacza to, że aby uruchomić nowy wątek, nie wystarczy jedynie stworzyć jego instancję — trzeba jeszcze określić kod, który ma być wykonywany w ramach tego wątku. Kod ten definiowany jest w metodzie `run()`, a samo rozpoczęcie działania następuje poprzez wywołanie metody `start()` na obiekcie typu `Thread`. Warto przy tym odróżnić: obiekt wątku (`Thread`) to reprezentacja wątku w kodzie (na poziomie języka Java), natomiast wątek jako jednostka wykonawcza to element zarządzany przez system operacyjny, który rzeczywiście wykonuje instrukcje programu. Utworzenie obiektu klasy `Thread` nie oznacza jeszcze, że nowy wątek systemowy został uruchomiony. Stanie się to dopiero po wywołaniu `start()`.

Aby utworzyć wątek, można utworzyć obiekt, który dziedziczy po klasie `Thread`. Takie podejście jest najbardziej podstawowe, natomiast wiąże się z ograniczeniem dziedziczenia. Drugą opcją jest implementacja interfejsu `Runnable` i przekazanie jej do obiektu `Thread`. Nie ogranicza to programisty, a dodatkowo pozwala na zastosowanie wielu dodatkowych rozwiązań, jak wyrażenia lambda. We współczesnych programach najczęściej korzysta się jednak z trzeciej opcji, jaką jest `ExecutorService`. `ExecutorService` tworzy pulę wątków, która wykorzystywana jest do wykonywania zadań. Jest to bardzo wygodne rozwiązanie, ponieważ system kolejkowania zadań, wykonywanie ich oraz zarządzanie dostępnością wątków jest zadaniem dla wyspecjalizowanego obiektu, a nie programisty, jak to wygląda w przypadku klasy `Thread` i użycia `Runnable`.

Java dostarcza klasy oraz metody, pozwalające zapewnić synchronizację i spójność danych:

- `synchronized`

  Słowo kluczowe `synchronized` służy do zapewnienia bezpieczeństwa współbieżnego dostępu do zasobów współdzielonych przez wiele wątków. Chroni sekcję krytyczną, czyli fragment kodu, który jest wykonywany przez jeden wątek naraz, aby uniknąć błędów wynikających z jednoczesnego dostępu do wspólnych danych. Gdy wątek wchodzi do bloku `synchronized`, uzyskuje tzw. monitor. W języku Java każdy obiekt może pełnić rolę monitora, co nazywane jest wewnętrzną blokadą. Oznacza to, że żaden inny wątek nie może wejść do bloku `synchronized` chronionego tym samym monitorem, dopóki pierwszy wątek nie opuści bloku. W ten sposób `synchronized` zapewnia wzajemne wykluczanie - podstawową właściwość ochrony sekcji krytycznych. Dodatkowo `synchronized` gwarantuje widoczność zmian w pamięci – wszystkie modyfikacje zmiennych dokonane wewnątrz bloku `synchronized` przez jeden wątek będą widoczne dla innych wątków, które później wejdą do bloku chronionego tym samym monitorem.

- `volatile`

  Słowo kluczowe `volatile` gwarantuje widoczność zmian zmiennej pomiędzy wątkami. Oznacza to, że jeśli jeden wątek zmieni wartość zmiennej oznaczonej jako `volatile`, to każdy inny wątek odczytujący tą zmienną zobaczy jej zaktualizowaną wartość.

- `wait()`

  Pozwala na wstrzymywanie działania wątku do momentu, gdy inny wątek wyśle sygnał do kontynuacji. Wątek oddaje monitor obiektu i przechodzi w stan oczekiwania, a gdy pojawi się sygnał, ponownie podejmuje próbę uzyskania dostępu do monitora i kontynuuje działanie.

- `notify() + notifyAll()`

  Metody `notify()` i `notifyAll()` służą do wybudzania wątków, które zostały wcześniej wstrzymane za pomocą metody `wait()`. `notify()` budzi jeden losowo wybrany wątek, natomiast `notifyAll()` wybudza wszystkie oczekujące wątki. Wybudzony wątek (lub wątki) nie przechodzi od razu do działania — musi najpierw ponownie uzyskać dostęp do monitora

- **Klasa** `Thread`

  Klasa `Thread` w Javie jest obiektową reprezentacją wątku, umożliwiającą jego utworzenie, uruchomienie i kontrolowanie w ramach programu.

- **Interfejs** `Lock`

  Interfejs `Lock` definiuje bardziej zaawansowany mechanizm synchronizacji niż `synchronized`, pozwalający na ręczne zarządzanie blokadą. Umożliwia on jawne przejmowanie i zwalnianie blokady za pomocą metod `lock()` oraz `unlock()`, oferując

większą elastyczność, możliwość próbnego, nieblokującego przejęcia blokady, a także wsparcie dla przerwań i oczekiwania z limitem czasu.

- **Klasy pozawalające na operacje atomowe**

  Są to klasy z pakietu `java.util.concurrent.atomic`, które zapewniają wykonywanie pojedynczych operacji na zmiennych w sposób atomowy, czyli nieprzerywalny i bezpieczny w środowisku wielowątkowym. Dzięki nim można przykładowo bezpiecznie inkrementować licznik, porównywać i wymieniać wartości bez konieczności stosowania blokad.

- **Klasy synchronizujące**

  Wyróżniamy między innymi `CountDownLatch` (licznik, który pozwala czekać aż inne wątki zakończą zadania), `CyclicBarrier` (wątki czekają, aż określona liczba osiągnie wspólny punkt), `Semaphore` (ogranicza liczbę jednoczesnych dostępów do zasobu), `BlockingQueue` (kolejka blokująca umożliwiająca bezpieczną wymianę danych między wątkami, z automatycznym oczekiwaniem przy próbie pobrania z pustej lub dodania do pełnej kolejki).

- `ExecutorService`

  `ExecutorService` to interfejs z pakietu `java.util.concurrent`, który definiuje zestaw metod do wykonywania asynchronicznych zadań. Umożliwia zgłaszanie zadań do wykonania, ich anulowanie oraz kontrolowanie cyklu życia wykonawcy. Konkretną obsługą puli wątków zajmują się jego implementacje, takie jak `ThreadPoolExecutor`, dzięki czemu programista nie musi ręcznie zarządzać wątkami.

Są to przykłady gotowych rozwiązań, których stosowanie wymaga znajomości ich dokładnego działania.

Rozdział 3

Java Agent

System monitorowania wątków w niniejszej pracy opiera się na Java agencie — specjalnym rodzaju aplikacji, która umożliwia monitorowanie wywołań metod w czasie działania programu. Aby w pełni zrozumieć działanie Thread Agenta, warto najpierw poznać podstawy dotyczące Java agentów.

3.1 Wstęp

Java agent jest programem, który umożliwia modyfikację aplikacji napisanych w języku Java na poziomie bajtowego kodu klas. Służy do instrumentacji kodu podczas działania aplikacji. Java agent może być statyczny i dynamiczny — statyczny jest ładowany podczas uruchamiania JVM (za pomocą opcji `-javaagent`), a dynamiczny może zostać załadowany do już działającej JVM w czasie działania programu (np. przez `VirtualMachine.attach()`).

Każdy program w Javie, aby został uruchomiony, potrzebuje klasy zawierającej metodę o odpowiedniej sygnaturze:

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

Listing 1: Sygnatura metody `main`

Statyczny Java agent, zamiast metody `main()`, powinien zawierać metodę o sygnaturze:

```
1 public static void premain(String agentArgs, Instrumentation inst)
```

Listing 2: Sygnatura metody `premain`

Kod statycznego Java agenta wykonywany jest, jak nazwa metody wskazuje, przed wykonaniem metody `main()`. Taka możliwość daje agentowi dostęp do klas i metod Javy oraz aplikacji użytkownika przed jej uruchomieniem. Aby uruchomić aplikację z Java agentem, należy dodać argument `-javaagent`, przykładowo:

```
java -javaagent:"/agent.jar" -jar app.jar
```

Listing 3: Uruchomienie aplikacji `app.jar` z Java agentem `agent.jar`

Przykładowy Java agent, który wypisuje informację o wersji JVM, systemie operacyjnym oraz czasie uruchomienia:

```
import java.lang.instrument.Instrumentation;

public class LoggingAgent {
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println("JVM version: " + System.getProperty("java.version"));
        System.out.println("OS: " + System.getProperty("os.name")
                + " " + System.getProperty("os.version"));
        System.out.println("Time: " + java.time.LocalDateTime.now());
    }
}
```

Listing 4: Kod źródłowy Java agenta wypisujący podstawowe informacje o systemie i aplikacji

W przypadku uruchomienia takiego agenta z aplikacją:

```
public class Example {
    public static void main(String[] args) {
        System.out.println("Running application...");
    }
}
```

Listing 5: Kod źródłowy prostej aplikacji Java

wynik będzie wyglądać następująco:

```
JVM version: 11.0.22
OS: Windows 11 10.0
Time: 2025-05-27T11:58:28.615180600
Running application...
```

Listing 6: Komunikaty wypisane na standardowe wyjście przez Java agenta i aplikację

Jest to bardzo prosty przykład działania Java agenta. Warto pamiętać, że aplikacja Java skompilowana daną wersją nie może zostać uruchomiona wersją poprzednią. Dotyczy to również Java agenta. Agent skompilowany wersją Java 17 nie będzie działał na wersji 11.

3.2 Interfejs `Instrumentation`

Główna metoda statycznego Java agenta przyjmuje dwa argumenty, `agentArgs` typu `String` oraz `inst` typu `Instrumentation`. Argument `agentArgs` zawiera parametry przekazane do Java agenta. Drugi argument typu `Instrumentation` pozwala na przechwytywanie, modyfikację oraz redefiniowanie klas w czasie działania programu. W takim obiekcie agent ma dostęp do wielu metod, w tym do metody `addTransformer()`, która przyjmuje obiekt typu `ClassFileTransformer`. `ClassFileTransformer` zawiera metodę `transform()` o sygnaturze:

```
byte[] transform(ClassLoader loader,
 String className,
 Class<?> classBeingRedefined,
 ProtectionDomain protectionDomain,
 byte[] classfileBuffer)
                  throws IllegalClassFormatException
```

Listing 7: Sygnatura metody `transform`

Ta metoda jest wywoływana za każdym razem, gdy JVM ładuje klasę. Przykładowo, żeby uzyskać komunikat o załadowaniu klasy z pakietu `org.example`, należy napisać takiego agenta:

```java
import java.lang.instrument.ClassFileTransformer;
import java.security.ProtectionDomain;

public class LoggingTransformer implements ClassFileTransformer {
    @Override
    public byte[] transform(
            ClassLoader loader,
            String className,
            Class<?> classBeingRedefined,
            ProtectionDomain protectionDomain,
            byte[] classfileBuffer) {

        if (className != null && className.startsWith("org/example")) {
            System.out.println(className + "is loaded");
        }

        return null;
    }
}
```

```java
import java.lang.instrument.Instrumentation;

public class LoggingAgent {
    public static void premain(String agentArgs, Instrumentation inst) {
        inst.addTransformer(new LoggingTransformer());
    }
}
```

Listing 8: Implementacja Java agenta, ktróry podczas ładowania klasy wypisuje jej nazwę, o ile należy ona do pakietu `org.example`

Przykładowy wynik w konsoli, w przypadku, gdy aplikacja użytkownika ma metodę `main()` w pakiecie `org.example`:

```
org/example/Main is loaded
```

Listing 9: Komunikat o załadowaniu klasy `org.example.Main`

W tym przykładzie metoda `transform()` zwraca pustą referencję, co oznacza, że transformacja nie modyfikuje zawartości klasy. Aby zrealizować zmianę, należy zwrócić nowy kod klasy w formie tablicy bajtów. Klasy w Javie są kompilowane do plików `.class`, które zawierają bajtkod — pośrednią formę kodu źródłowego Javy, interpretowaną przez wirtualną maszynę Java (JVM). Własnoręczna modyfikacja tego kodu jest trudna, dlatego do takich zadań wykorzystuje się gotowe narzędzia.

3.3 ASM

ASM[9] to niskopoziomowa biblioteka umożliwiająca odczyt, analizę oraz modyfikację bajtkodu Javy. Działa bezpośrednio na strukturach klas zapisanych w formacie `.class`, zgodnym ze specyfikacją JVM. Dzięki temu pozwala programistom tworzyć nowe klasy w locie, ingerować w istniejące metody lub rozszerzać ich funkcjonalność bez konieczności ponownej kompilacji źródeł. Działa w oparciu o wzorzec `Visitor` – klasy są odczytywane za pomocą `ClassReader`, przetwarzane za pomocą `ClassVisitor` i `MethodVisitor`, a następnie zapisywane z użyciem `ClassWriter`. ASM znajduje zastosowanie w programowaniu aspektowym, generowaniu klas przez frameworki, optymalizacji kodu w czasie kompilacji lub uruchamiania oraz w narzędziach deweloperskich. Na tle bibliotek wyższego poziomu ASM wyróżnia się:

- większą wydajnością
- bezpośrednim dostępem do instrukcji bajtkodu

Biblioteka ASM jest szczególnie przydatna w sytuacjach, gdy wymagana jest pełna kontrola nad strukturą klasy i maksymalna wydajność działania. Jest ona również podstawowym komponentem wykorzystywanym wewnętrznie przez inne narzędzia.

Thread Agent wykorzystuje ASM przy monitorowaniu bloków `synchronized`.

3.4 Biblioteka Byte Buddy

Byte Buddy jest biblioteką, na której oparty jest Thread Agent.

3.4.1 Wstęp

Byte Buddy[10] to nowoczesna biblioteka, która służy do generowania i modyfikowania kodu Javy. Byte Buddy modyfikuje kod bajtowy, dzięki czemu nie wymaga kompilatora do wdrożenia zmian. Pozwala ona na dynamiczne tworzenie, modyfikowanie i instrumentowanie klas podczas działania aplikacji. Dzięki temu możliwe jest wpływanie na zachowanie programu bez konieczności modyfikowania jego kodu źródłowego. Byte Buddy jest szczególnie popularny w Java agentach.

3.4.2 Zastosowanie

Byte Buddy powstał jako wysokopoziomowa alternatywa dla niskopoziomowych bibliotek takich jak ASM czy Javassist[11], które również umożliwiają manipulacje bajtkodem, ale wymagają bardzo dobrej znajomości struktury JVM. Byte Buddy upraszcza ten proces, udostępniając przejrzyste, obiektowe API. Dzięki temu można w prosty sposób:

- tworzyć klasy w locie,
- zmieniać istniejące metody (np. dodawać logowanie przed i po ich wywołaniu),
- dodawać nowe metody lub pola do klas,

- delegować zachowania do innych klas,
- monitorować wywołania metod w istniejących aplikacjach.

Byte Buddy znajduje zastosowanie w wielu scenariuszach, w tym w instrumentacji klas systemowych w celu dodania logowania, pomiaru czasu wykonania lub analizy działania aplikacji bez modyfikowania jej kodu źródłowego. Jednym z najważniejszych zastosowań jest współpraca z mechanizmem Java agentów. Byte Buddy wspiera proces modyfikacji bajtkodu poprzez klasę `AgentBuilder`, która pozwala łatwo zdefiniować, które klasy mają być instrumentowane i w jaki sposób. Dzięki temu możliwe jest przykładowo monitorowanie wszystkich metod w klasach użytkownika, a także metod z pakietu `java.lang`.

Zalety Byte Buddy:

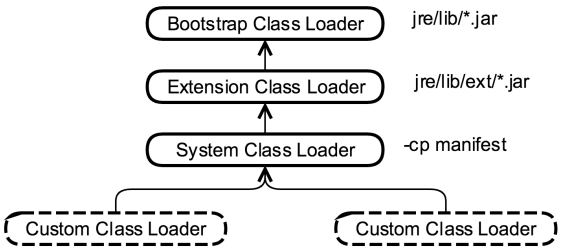
- Czytelność: API jest proste, intuicyjne i łatwe do zrozumienia.
- Bezpieczeństwo: generowany kod jest w pełni zgodny z wymogami i restrykcjami JVM, co zapewnia stabilność i integralność aplikacji.
- Integracja z agentami: pełne wsparcie dla Java Instrumentation API.
- Brak konieczności poznania kodu źródłowego: można modyfikować klasy bez posiadania wiedzy na temat aplikacji użytkownika.
- Dynamiczność: pozwala reagować na warunki środowiskowe i dostosowywać zachowanie programu w czasie jego działania.

3.4.3 Podsumowanie

Byte Buddy to potężne narzędzie do manipulacji klasami w czasie działania programu, które znacznie ułatwia tworzenie rozwiązań wymagających dynamicznej instrumentacji. Dzięki przystępnemu API oraz dobrej integracji z Java agentami, biblioteka ta znajduje zastosowanie w zaawansowanych narzędziach programistycznych, środowiskach testowych, jak również w produkcyjnych systemach wymagających dynamicznego monitorowania działania aplikacji.

3.5 ClassLoader

ClassLoadery w Java agentach powodują bardzo poważne problemy. Aby zrozumieć powody tych problemów, należy wytłumaczyć zasadę działania ładowarek klas. W Javie każda klasa jest ładowana przez obiekt zwany ClassLoader. Klasa ta zajmuje się znalezieniem i wczytaniem pliku `.class`. Każda klasa ładowana jest dokładnie raz na jedno uruchomienie programu. Dzieje się to w momencie pierwszego użycia danej klasy. Przykładowo w momencie wykonywania metody `premain()`, klasa z metodą `main()` aplikacji użytkownika nie jest załadowana, ale klasa `Thread` już jest, ponieważ wymagana była do stworzenia wątku, który wykonuje kod Java agenta. Klasy ładowane są według konkretnego algorytmu. Wyróżnia się trzy główne ClassLoadery, które tworzą hierarchię (Rysunek 3.1).



Rysunek 3.1: Hierarchia ClassLoaderów[14]

ClassLoader	**Opis**
Bootstrap ClassLoader	Ładuje najważniejsze i najbardziej podstawowe klasy Javy, takie jak te z pakietów `java.lang`, `java.util`, `java.io`.
Platform ClassLoader	Ładuje klasy rozszerzeń platformy, m.in. z modułów `java.xml`, `java.sql`, `java.desktop`. Zastępuje Extension ClassLoader od JDK 9.
Application ClassLoader	Ładuje klasy aplikacji użytkownika, czyli te znajdujące się na ścieżce `classpath`, np. z katalogów projektu lub plików JAR.

Tabela 3.1: Przegląd głównych ClassLoaderów w JVM

ClassLoader stosuje model delegacji: najpierw przekazuje żądanie załadowania klasy do swojego rodzica. Jeśli ten jej nie znajdzie, ClassLoader próbuje załadować klasę samodzielnie. Każdy ClassLoader jest odpowiedzialny za ładowanie określonego zestawu klas (Tabela 3.1).

Problemy pojawiają się w momencie, gdy programista zaczyna się zastanawiać, czy Java agent jest częścią aplikacji użytkownika. Teoretycznie tak – działa w ramach tego samego procesu wirtualnej maszyny Javy. W praktyce jednak agent powinien być traktowany jako odrębny komponent. Implementacja Java agenta wymaga dużej ostrożności, zwłaszcza gdy ingeruje on w klasy użytkownika lub korzysta z zewnętrznych bibliotek. Klasy agenta są domyślnie ładowane przez SystemClassLoader, czyli w ten sam sposób, co klasy aplikacji. Powoduje to, że aplikacja ma dostęp do klas agenta, co jest zazwyczaj niepożądane. Ponieważ agent zawiera niezależną logikę, współdzielenie klas może prowadzić do konfliktów — może dojść do nadpisania lub niezamierzonego użycia klas pomocniczych bądź bibliotek załadowanych przez agenta przez aplikację użytkownika, jak również do sytuacji odwrotnej, w której agent korzysta z klas aplikacji o identycznych nazwach pakietów lub różnych wersjach tej samej biblioteki. Jeśli agent załaduje daną bibliotekę jako pierwszy, aplikacja użytkownika nie będzie mogła załadować innej wersji tej samej biblioteki, co skutkuje błędami przy uruchomieniu. Aby uniknąć tego typu problemów, zaleca się tworzenie dedykowanego ClassLoadera dla Java agenta. Taki loader powinien być niewidoczny dla aplikacji użytkownika i odpowiedzialny wyłącznie za ładowanie klas związanych z agentem. Pozwala to na pełną separację logiki agenta od aplikacji, minimalizując ryzyko kolizji i niepożądanych interakcji.

Rozdział 4

Thread Agent

4.1 Przedstawienie aplikacji

Thread Agent jest aplikacją, która pozwala monitorować wątki poprzez analizę wywołań metod. Modyfikuje bajtkod aplikacji użytkownika, aby dodać logowanie przydatnych informacji naokoło metod odpowiedzialnych za synchronizację i cykl życia wątków.

4.2 Opis działania aplikacji

4.2.1 Monitorowanie wywołań metod

Thread Agent to narzędzie służące do monitorowania środowiska wielowątkowego w aplikacjach opartych na platformie Java. Umożliwia ono obserwację i analizę działania wątków, w tym ich tworzenia, uruchamiania, kończenia oraz interakcji z innymi wątkami. Aplikacja monitoruje następujące operacje:

- Stworzenie obiektu klasy `Thread`

  Podczas tworzenia wątku przez JVM logowana jest informacja zawierająca jego nazwę. Dodatkowo rejestrowane jest, czy wątek został utworzony za pośrednictwem `ExecutorService`, czy też niezależnie. Przykładowo:

```
2025-06-02 17:18:04 INFO  ThreadConstructorAdvice - Thread Thread-5 created independently
2025-06-02 17:18:04 INFO  ThreadConstructorAdvice - Thread pool-3-thread-1 created by Executor:
↪  java.util.concurrent.ThreadPoolExecutor@3f51ee4f[Running, pool size = 0, active threads = 0, queued
↪  tasks = 0, completed tasks = 0]
```

Listing 10: Logi Thread Agenta wskazujące na utworzenie nowych wątków

- Wywołania metod `Lock.lock()`, `Lock.unlock()`

  Thread Agent monitoruje metody `lock()` i `unlock()` wykonane na obiekcie `Lock`. Wyświetla informację o wątku, który wykonał metodę, wraz z miejscem w kodzie aplikacji oraz szczegółami. Przykładowo:

```
2025-06-02 17:18:04 INFO  LockAdvice - Lock java.util.concurrent.locks.ReentrantLock@76721208[Unlocked]
↪  acquired (or waiting to be acquired) by thread main at org.example.Main.main(Main.java:27)
2025-06-02 17:18:04 INFO  UnlockAdvice - Lock java.util.concurrent.locks.ReentrantLock@76721208[Locked by
↪  thread main] released by thread main at org.example.Main.main(Main.java:28)
```

Listing 11: Logi Thread Agenta wskazujące na przejęcie i zwolnienie blokady

- Zamknięcie egzekutora

  Thread Agent monitoruje metodę `shutdown()` wykonaną na obiekcie `ExecutorService`. Wyświetla szczegółowe informacje o obiekcie, przykładowo

```
2025-06-02 17:18:04 INFO  ExecutorShutdownAdvice - Executor
↪  java.util.concurrent.ThreadPoolExecutor@29f20756[Running, pool size = 1, active threads = 1, queued
↪  tasks = 0, completed tasks = 1] shutdown
```

Listing 12: Logi Thread Agenta wskazujące na zamknięcie egzekutora

- Wywołanie metody `Thread.start()`

  Thread Agent monitoruje rozpoczęcie wykonywania pracy wątku, przykładowo:

```
2025-06-02 17:18:04 INFO  ThreadStartAdvice - Started new thread - Thread[#48,Thread-4,5,main]
```

Listing 13: Logi Thread Agenta wskazujące na rozpoczęcie pracy wątku

- Stworzenie obiektu klasy `ExecutorService`

  Thread Agent zaloguje informację, jeżeli zostanie utworzony `ExecutorService`. Zostanie podana również informacja o miejscu w kodzie aplikacji, w którym został `ExecutorService` utworzony, przykładowo:

```
2025-06-03 22:05:15 INFO  ExecutorConstructorAdvice - Thread main created new
↪  java.util.concurrent.ThreadPoolExecutor@79607189[Running, pool size = 0, active threads = 0, queued
↪  tasks = 0, completed tasks = 0] at org.example.Main.main(Main.java:75)
```

Listing 14: Logi Thread Agenta wskazujące na utworzenie `ExecutorService`

- Wywołania metod `Executor.execute()`, `ExecutorService.submit()`

  Thread Agent monitoruje moment uruchomienia zadania w `ExecutorService`, wskazując nazwę wątku wykonującego oraz dokładne miejsce wywołania `submit()` lub `execute()` w kodzie aplikacji, przykładowo:

```
2025-06-02 17:18:04 INFO  ExecutorExecuteSubmitAdvice - Task executed by thread <main> at
↪  org.example.Main.main(Main.java:98) on
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@60fa2d75
2025-06-02 17:18:04 INFO  ExecutorExecuteSubmitAdvice - Task submitted by thread <main> at
↪  org.example.Main.main(Main.java:87) on java.util.concurrent.ThreadPoolExecutor@3f51ee4f[Running, pool
↪  size = 0, active threads = 0, queued tasks = 0, completed tasks = 0]
```

Listing 15: Logi Thread Agenta wskazujące na wprowadzenie zadania do `ExecutorService`

- Wywołanie metody `Thread.sleep()`

  Thread Agent wychwytuje moment uśpienia wątku. Loguje informację o czasie rozpoczęcia i zakończenia metody `sleep()`, przykładowo:

```
2025-06-02 17:18:04 INFO  SleepSubstitution - Thread Background Thread started sleeping for 1000 ms
2025-06-02 17:18:05 INFO  SleepSubstitution - Thread Background Thread finished sleeping for 1000 ms
```

Listing 16: Logi Thread Agenta wskazujące na uśpienie wątku

- Wywołania metod `Object.notify()`, `Object.notifyAll()`

  Thread Agent zapisuje informacje o wątku, który wykonał metodę `notify()` oraz metodę `notifyAll()`, przykładowo

```
2025-06-03 22:05:17 INFO  NotifyAllSubstitution - Thread Thread-4 called notifyAll() on object:
↪  org.example.Main@160eb1e5
2025-06-03 22:05:17 INFO  WaitSubstitution - Thread Thread-2 got notified on object:
↪  org.example.Main@160eb1e5
2025-06-03 22:05:17 INFO  WaitSubstitution - Thread Thread-3 got notified on object:
↪  org.example.Main@160eb1e5
```

Listing 17: Logi wskazujące, że wątek `Thread-4` wywołał `notifyAll()`, budząc oczekujące wątki `Thread-2` i `Thread-3` na monitorze obiektu `Main`

- Wywołanie metody `Object.wait()`

  Uśpienie wątku za pomocą metody `wait()` zostanie odnotowane wraz z informacją o obiekcie, na którym wykonana zostanie metoda `wait()`. przykładowo:

```
2025-06-03 22:05:15 INFO  WaitSubstitution - Thread Thread-3 called wait() on object:
↪  org.example.Main@160eb1e5
2025-06-03 22:05:17 INFO  WaitSubstitution - Thread Thread-3 got notified on object:
↪  org.example.Main@160eb1e5
```

Listing 18: Logi Thread Agenta pokazujące moment uśpienia wątku za pomocą `wait()` i następnie jego wznowienia

- Wywołanie metody `Object.yield()`

  Thread Agent loguje informację o wywołaniu metody `yield()`, przykładowo:

```
2025-06-03 22:05:15 INFO  YieldSubstitution - Thread main called yield()
```

Listing 19: Logi Thread Agenta wskazujące na wywołanie metody `yield`

- Wywołania metod z interfejsu `Condition`: `signal()`, `signalAll()`, `await()`, `awaitUntil()`, `awaitNanos()`, `awaitUninterruptibly()`

```
2025-09-03 14:06:52 INFO  ConditionSubstitution - Thread Thread-3 awaits on
↪  java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@297ec973
2025-09-03 14:06:53 INFO  ConditionSubstitution - Thread Thread-4 wakes up one thread on
↪  java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@297ec973
2025-09-03 14:06:53 INFO  ConditionSubstitution - Thread Thread-3 woken up
```

Listing 20: Logi agenta wskazujące na moment rozpoczęcia oczekiwania (wywołanie `await()`), moment wysłania sygnału (`signal()`) oraz chwilę zakończenia oczekiwania i wznowienia działania wątku

- Wywołania bloków `synchronized`

  Thread Agent rejestruje zdarzenia związane z oczekiwaniem na wejście do bloku `synchronized`, momentem wejścia oraz momentem jego opuszczenia.

```
2025-09-02 12:43:53 INFO  SynchronizedLogger - Thread main waiting to hold monitor: class org.example.Main
2025-09-02 12:43:53 INFO  SynchronizedLogger - Thread main holds monitor: class org.example.Main
2025-09-02 12:43:53 INFO  SynchronizedLogger - Thread main released monitor: class org.example.Main
```

Listing 21: Logi wskazujące na oczekiwanie, wejście oraz wyjście z bloku `synchroznied`

- Wywołania metod `synchronized`

  Thread Agent rejestruje zdarzenia związane z wejściem oraz wyjściem z metody `synchronized`.

```
2025-09-02 12:43:53 INFO  SynchronizedLogger - Thread main holds monitor: class org.example.Main
2025-09-02 12:43:53 INFO  SynchronizedLogger - Thread main released monitor: class org.example.Main
```

Listing 22: Logi wskazujące na wejście oraz wyjście z metody synchroznied

Thread Agent monitoruje zarówno metody, jak i bloki oznaczone słowem kluczowym `synchronized` w kodzie bajtowym Javy. Dla bloków rejestrowane są trzy zdarzenia: oczekiwanie na wejście, moment wejścia i moment wyjścia. W przypadku metod `synchronized` domyślnie rejestrowane są jedynie moment wejścia oraz wyjścia (bez fazy oczekiwania). Aby pozwolić użytkownikowi monitorować również moment oczekiwania na wejście do metody `synchronized` Thread Agent udostępnia moduł `monitoring-call`. Moduł dostarcza klasy,

które zawierają puste (bez ciała) metody, które są przechwytywane przez Thread Agenta i służą między innymi do sygnalizowania oczekiwania na wejście do metody `synchronized`. Dzięki temu monitorowanie metod `synchronized` może — analogicznie do bloków — obejmować również fazę oczekiwania, gdy jest to potrzebne.

- `org.threadmonitoring.call.SynchronizedCall#alertBeforeSynchronizedEntry` Monitoruje chęć wejścia do bloku `synchronized`. Przykładowy log:

```
2025-09-02 12:43:53 INFO  SynchronizedLogger - Thread main waiting to hold monitor: class org.example.Main
```

Listing 23: Logi Thread Agenta wskazujące na wątek, który oczekuje na wejście do bloku `synchronized`

Przykładowe użycie metody:

```
synchronized static void synchronizedMethod() {
...
}
public static void main(String[] args) {
    SynchronizedCall.alertBeforeSynchronizedMethodEntry(Main.class);
    synchronizedMethod();
}
```

Listing 24: Kod programu, który korzystając z biblioteki `monitoring-call` dodaje monitorowanie oczekiwania na wejście do bloku `synchronized`

Thread Agent nie monitoruje automatycznie wszystkich metod związanych ze współbieżnością. Użytkownik może chcieć śledzić również własne metody lub rejestrować momenty, w których wątki przechodzą przez określone fragmenty kodu. Aby to umożliwić, Thread Agent udostępnia dodatkową klasę `org.threadmonitoring.call.Call` w module `monitoring-call`. Klasa ta zawiera jedną metodę (bez implementacji), której wywołania są przechwytywane i odpowiednio logowane przez Thread Agenta. Przykładowo, podczas monitorowania takiego bloku kodu:

```
Lock l = new ReentrantLock();
Call.alertMultithreadingCall("RandomClass.randomMethod();");
RandomClass.randomMethod();
```

Listing 25: Kod programu, który korzystając z biblioteki `monitoring-call` dodaje monitorowanie wywołania metody `RandomClass.randomMethod()`

zostanie zalogowana informacja:

```
2025-06-03 21:52:14 INFO  GeneralSubstitution - Thread main called RandomClass.randomMethod();
```

Listing 26: Logi Thread Agenta wskazujące na wywołanie metody `RandomClass.randomMethod()`

4.2.2 Monitorowanie stanu wąków przy zamknięciu JVM

W momencie zakończenia działania JVM, jeśli istnieją wątki w stanie innym niż `TERMINATED`, zostanie zalogowana informacja o każdym takim wątku wraz ze ścieżką wywołań w momencie przerwania.

```
Thread Thread-3 was forced to terminate during JVM shutdown with state: TIMED_WAITING
Stacktrace:
        at java.base/java.lang.Thread.sleep0(Native Method)
        at java.base/java.lang.Thread.sleep(Thread.java:509)
        at org.threadmonitoring.substitution.SleepSubstitution.sleep2(SleepSubstitution.java:12)
        at org.example.Main.lambda$main$0(Main.java:7)
        at java.base/java.lang.Thread.run(Thread.java:1583)
```

Listing 27: Logi Thread Agenta wskazujące na niezakończony wątek `Thread-3`

4.2.3 Analiza wywołań

Thread Agent analizuje wywołania metod w czasie rzeczywistym, wykrywając potencjalne problemy w aplikacji i zapisując je do dedykowanego pliku `thread-agent_emergency.log`, obok standardowego pliku `thread-agent.log`. Aplikacja rejestruje m.in.:

- **Potencjalne zakleszczenia wątków**

```
Object a = new Object();
Object b = new Object();
new Thread(() -> {
    synchronized(a) {
        try { Thread.sleep(100); } catch (InterruptedException ignored) {}
        synchronized(b) {}
    }
}).start();

new Thread(() -> {
    synchronized(b) {
        try { Thread.sleep(100); } catch (InterruptedException ignored) {}
        synchronized(a) {}
    }
}).start();
```

Listing 28: Kod, który prowadzi do zakleszczenia wątków

```
[2025-09-01 20:50:42] Potential deadlock detected!
[2025-09-01 20:50:42] Deadlock cycle:
[2025-09-01 20:50:42] Thread[#48,Thread-4,5,main] -> Thread[#47,Thread-3,5,main] -> Thread[#48,Thread-4,5,main]
[2025-09-01 20:50:42] Stack traces of involved threads:
[2025-09-01 20:50:42] Thread: Thread-4 (ID: 48)
[2025-09-01 20:50:42]     at org.example.Main.lambda$main$1(Main.java:18)
[2025-09-01 20:50:42]
[2025-09-01 20:50:42] Thread: Thread-3 (ID: 47)
[2025-09-01 20:50:42]     at org.example.Main.lambda$main$0(Main.java:11)
```

Listing 29: Log wygenerowany po wykryciu zakleszczenia wątków

- **Wywołania metod `notify`/`wait` na obiekcie bez posiadania na nim monitora**

```
class Main {
    public static void main(String[] args) {
        Main.class.notify();
    }
}
```

Listing 30: Kod powodujący wyrzucenie wyjątku IllegalMonitorStateException

```
[2025-09-01 20:51:48] Thread main called notify() on object: class org.example.Main without holding the monitor
↪  on required object! Application will throw IllegalMonitorStateException
```

Listing 31: Log wygenerowany w przypadku wywołania metody `notify()` bez posiadania monitora na obiekcie

- **Przerwanie stanu uśpienia wątku**

```
Thread t = new Thread(() -> {
    try { Thread.sleep(1000); }
    catch (InterruptedException ignored) { }
});
t.start();
try { Thread.sleep(300); }
catch (InterruptedException ignored) { }
t.interrupt();
```

Listing 32: Kod powodujący przerwanie stanu uśpienia wątku

```
[2025-09-01 20:55:01] Thread Thread[#1,main,5,main] is interrupting thread Thread[#47,Thread-3,5,main] in
↪  TIMED_WAITING state! There is a possibility to throw InterruptedException!
```

Listing 33: Log wygenerowany po wykryciu przerwania stanu uśpienia wątku

4.3 Uruchomienie i konfiguracja aplikacji

Aby uruchomić Thread Agenta, należy umieścić podane projekty[15] w jednym folderze:

- **Thread Agent API**
- **Thread Agent Entry**
- **Monitoring call**
- **Thread Agent**

Następnie zbudować każdy z nich według podanej powyżej kolejności instrukcją

```
./gradlew jar
```

Listing 34: Instrukcja do zbudowania projektu

Alternatywą jest użycie zbudowanego Thread Agenta[16].

Następnie należy skonfigurować agenta. Aby to zrobić, należy otworzyć w dowolnym edytorze plik `threadmonitoring\configuration\conf.yml`, a następnie wypisać wszystkie pakiety, które mają być monitorowane przez Thread Agenta. W przypadku chęci monitorowania całej aplikacji, należy zostawić klasy `java.lang.Thread` oraz `java.lang.Object` i dopisać początek wspólnego pakietu dla pozostałych klas. Aplikacja, monitorując metody, generuje pliki, w których loguje wszystkie wydarzenia. Miejsce i format logów można ustawić w pliku `log4j2.xml`.

Aby uruchomić monitorowanie aplikacji, należy podać parametr

```
-javaagent:"<Ścieżka absolutna do pliku thread-agent.jar>"
```

Listing 35: Parametr wstrzykujący Thread Agenta do aplikacji

podczas uruchamiania docelowego pliku `.jar`. Przykładowo:

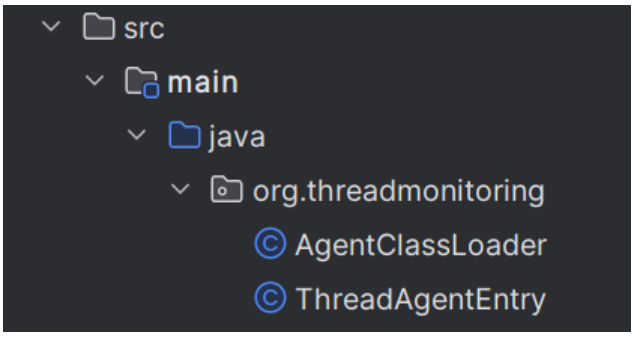
```
java -javaagent:"/threadmonitoring/thread-agent.jar" -jar my-app.jar
```

Listing 36: Uruchomienie aplikacji z Thread Agentem

Dodatkowo, w przypadku chęci zmiany domyślnego folderu, do którego Thread Agent loguje zdarzenia, można dodać parametr

```
log4j2.logdir
```

Listing 37: Parametr zmieniający domyślny folder do logowania



Rysunek 4.1: Struktura klas w Thread Agent Entry

Thread Agent wspiera wszystkie aplikacje napisane w języku Java od wersji 11. Po poprawnym uruchomieniu, aplikacja powinna wypisać logi na standardowe wyjście potwierdzające poprawne uruchomienie Thread Agenta:

```
> java -javaagent:"C:\Users\Piotr\thread-agent.jar" -jar .\my-app.jar
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes
↪  because bootstrap classpath has been appended
Initializing ThreadAgent before the target application to enable thread and
↪  executor monitoring
The logging has been configured to the "C:\Users\Piotr/logs
Initialized advices
Advices and method substitutions created and installed
Attempting to retransform classes
Retransformation completed successfully
Transformation and Retransformation finished, Thread Agent is working, running
↪  target application...
```

Listing 38: Logi wskazujące na poprawnie uruchomienie Thread Agenta

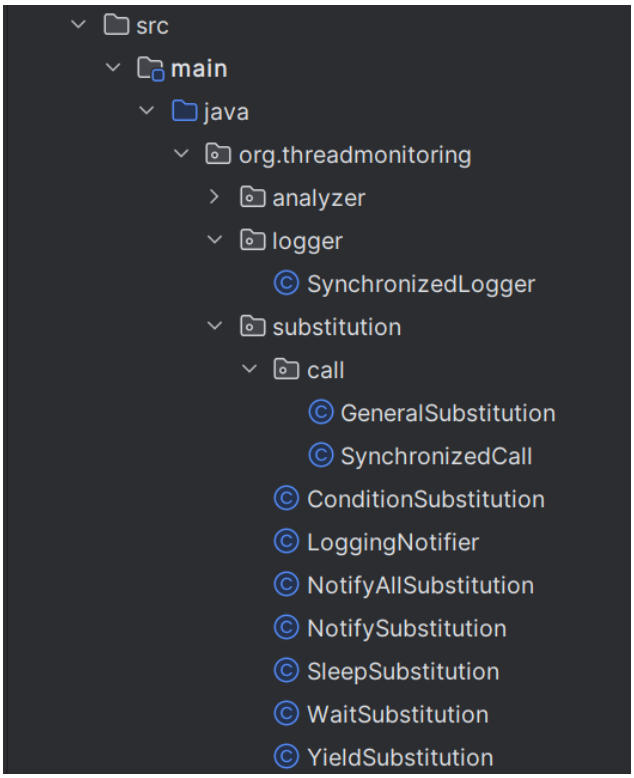
Taka informacja oznacza, że Thread Agent został uruchomiony poprawnie. Logi można sprawdzić pod ścieżką, która jest widoczna na standardowym wyjściu. W wygenerowanych logach można znaleźć informacje przydatne do rozwiązywania problemów wielowątkowych. Interpretacja logów na praktycznych przykładach znajduje się w rozdziale szóstym.

4.4 Techniczny opis aplikacji

Thread Agent jest statycznym Java agentem, który monitoruje aplikacje użytkownika. Korzysta z biblioteki Byte Buddy do instrumentacji kodu, `log4j` do logowania oraz Jackson[13] do parsowania konfiguracji. System składa się z kilku modułów:

1. **Thread Agent Entry**

   Thread Agent Entry to początkowy plik `.jar`, który zawiera klasę z metodą `premain()`. Odpowiedzialny jest za załadowanie publicznego API agenta tak, aby

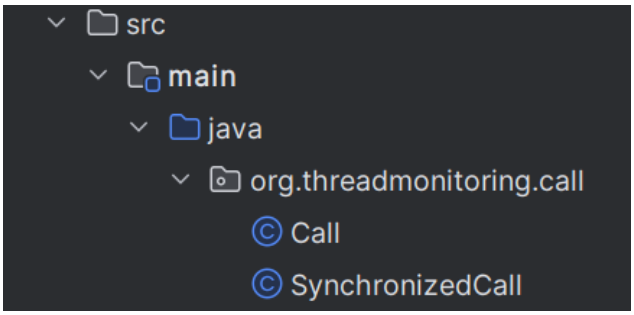


Rysunek 4.2: Struktura klas w Thread Agent API

był dostępny dla aplikacji użytkownika oraz dla Thread Agenta. Entry posiada klasę `AgentClassLoader`, która jest własnym ClassLoaderem odpowiedzialną za ładowanie klas w Thread Agencie tak, aby wewnętrzne klasy i metody aplikacji monitorującej nie były widoczne dla aplikacji użytkownika. `AgentClassLoader` jest wykorzystany w tym komponencie do załadowania wszystkich bibliotek używanych w agencie, a także ładuje nim klasę głównego komponentu Thread Agenta. Warto zaznaczyć, że jeżeli dana klasa została załadowana przez własny ClassLoader, a ten ClassLoader został ustawiony jako kontekstowy (context ClassLoader), to wszystkie klasy ładowane podczas wykonywania kodu tej klasy będą również ładowane przez ten sam ClassLoader. Context ClassLoader to specjalny mechanizm w Javie, który umożliwia bibliotekom ładowanie klas w kontekście środowiska uruchomieniowego — jest to ClassLoader przypisany do bieżącego wątku i może zostać ustawiony za pomocą metody `Thread.setContextClassLoader(ClassLoader)`.

2. **Thread Agent API**

   Thread Agent API to biblioteka współdzielona pomiędzy aplikacją użytkownika, a agentem, której zadaniem jest udostępnienie metod wywoływanych po stronie użytkownika. Zawiera nadpisane wersje metod natywnych z dodaniem monitorowania, implementacje obsługujące śledzenie bloków `synchronized` i dodatkowe punkty wywołań stworzone przez użytkownika.



Rysunek 4.3: Struktura klas w Monitoring Call

3. **Monitoring call**

   Monitoring Call to projekt zawierający klasy umożliwiające użytkownikowi oznaczanie dodatkowych miejsc w kodzie aplikacji, które mają być rejestrowane przez Thread Agenta. Umożliwia oznaczanie oczekiwania na wejście do metody `synchronized` oraz metod, które nie są domyślnie monitorowane przez agenta.

4. **Thread Agent**

   Thread Agent to główny komponent aplikacji, który jest odpowiedzialny za:

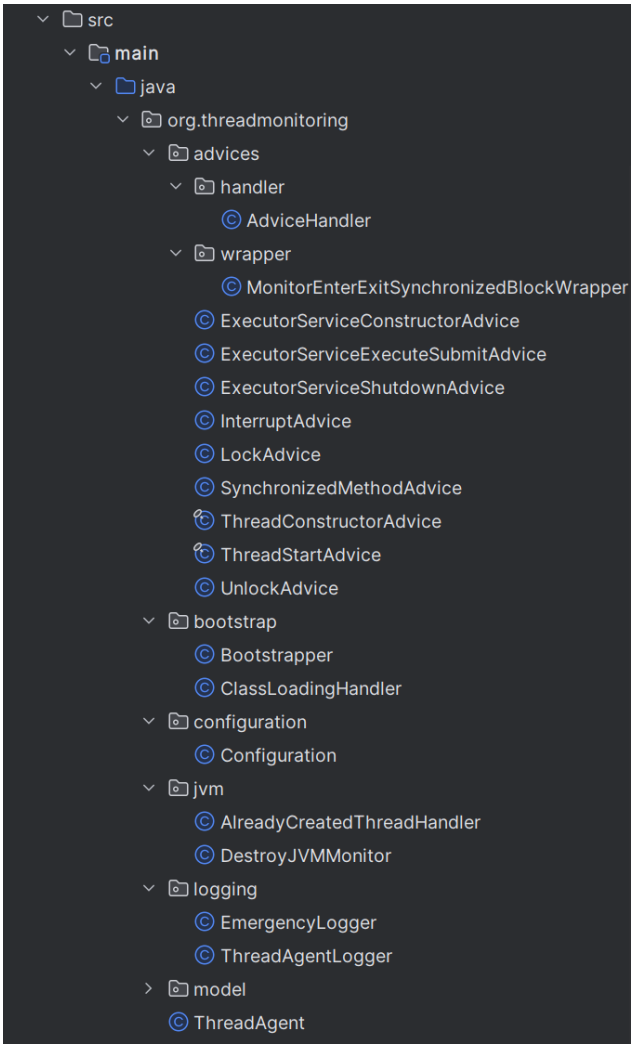
   (a) Inicjalizację `log4j`
   Najpierw Thread Agent zaczytuje parametr `log4j2.logdir` i na podstawie jego obecności podczas uruchomienia aplikacji, inicjalizuje `log4j`.

   (b) Wczytanie konfiguracji użytkownika
   Zaczytanie konfiguracji z pliku `conf.yml`, aby monitorować tylko wskazane przez użytkownika pakiety.

   (c) Inicjalizację kolejek logujących wydarzenia
   Niektóre wydarzenia wywoływane są poza Thread Agentem (w aplikacji użytkownika, w `thread-agent-api`). Aby uniknąć wielu różnych konfiguracji logowania, w Thread Agencie znajduje się kolejki, do których różne komponenty umieszczają informacje do zapisania w plikach dziennika.

   (d) Obsługę ładowania klas w tym użycie bootstrap method
   Thread Agent korzysta z zaawansowanego mechanizmu dynamicznego wiązania metod, aby umożliwić klasom ładowanym przez inne ClassLoadery (niż ten używany przez agenta) dostęp do interceptorów. Interceptor to komponent, który przechwytuje wywołania metod w celu ich monitorowania, modyfikacji lub rozszerzenia. Rozwiązanie to jest inspirowane rozwiązaniem z `apm-agent-java`[17]. Aby dokładnie zrozumieć takie działanie, należy wprowadzić kilka definicji:

   - **MethodHandle** Jest to uchwyt do metody, konstruktora lub innego punktu wywołania.
   - **invokedynamic** Instrukcja bytecode, która dynamicznie wiąże metodę podczas działania programu.



Rysunek 4.4: Struktura klas w Thread Agent

- **metoda bootstrap** Specjalna metoda, która jest wywoływana przez JVM przy pierwszym wywołaniu instrukcji `invokedynamic`.
- **CallSite** abstrakcja, która reprezentuje miejsce wywołania metody dynamicznej oraz powiązanie między `invokedynamic`, a konkretnym wywołaniem `MethodHandle`, do którego posiada referencję.

Thread Agent za pomocą metody bootstrap dynamicznie wstrzykuje i wywołuje metody monitorujące kod. Jest to zrobione w taki sposób, ponieważ domyślnie klasy ładowane przez Bootstrap ClassLoader (Bootstrap ClassLoader jest oznaczany jako pusta referencja na ClassLoader) nie mają dostępu do klas ładowanych innymi ClassLoaderami. Z powodu monitorowania klas Javy, takich jak `java.lang.Thread`, należy znaleźć obejście, które pozwala takim klasom na wywołanie metod logujących. Rozwiązaniem jest użycie metody bootstrap, która pozwala na dostęp klasom ładowanym przez Bootstrap ClassLoader do metod monitorujących, dodatkowo zabezpieczając się przed dostępem do nich klasom użytkownika.

(e) Stworzenie interceptorów
Thread Agent bazuje na interceptorach, które tworzy, aby przechwytywać metody dodając do nich logowanie i monitoring. Thread Agent nie zmienia działania metod.

(f) Transformację oraz retransformację klas
Transformacja to proces modyfikowania kodu bajtowego klasy podczas jej pierwszego załadowania przez ClassLoader. Retransformacja jest zmianą kodu bajtowego załadowanej już klasy. Thread Agent wykonuje zarówno transformację (przykładowo klas użytkownika), jak i retransformację (przykładowo klasy `java.lang.Thread`).

(g) Ustawienie odpowiednich ClassLoaderów dla aplikacji
Thread Agent przed uruchomieniem docelowej aplikacji ustawia domyślnego ClassLoadera oraz kontekst ClassLoadera, aby działanie agenta nie wpłynęło na działanie aplikacji użytkownika.

4.5 Wykorzystywanie biblioteki Byte Buddy

Biblioteka Byte Buddy odgrywa kluczową rolę w działaniu Thread Agenta, który intensywnie z niej korzysta do monitorowania metod w środowisku wielowątkowym. Aby śledzić wywołania metod, konieczna jest modyfikacja ich bajtkodu. Thread Agent wykorzystuje klasę `net.bytebuddy.agent.builder.AgentBuilder` do skonfigurowania agenta. Tworzony obiekt `AgentBuilder` otrzymuje szczegóły dotyczące transformacji i ewentualnej retransformacji klas ładowanych przez JVM za pomocą przygotowanej wcześniej konfiguracji.

W dalszej kolejności agent, w zależności od tego, czy dana metoda zawiera bajtkod (czyli nie jest metodą natywną), wykorzystuje gotowe, wysokopoziomowe mechanizmy dostarczane przez Byte Buddy — takie jak interceptory (Advice, MethodDelegation) oraz techniki podstawiania kodu (Substitution). Na podstawie tych reguł Byte Buddy generuje odpowiednie transformery, które są rejestrowane w maszynie wirtualnej za pomocą interfejsu Instrumentation. W efekcie

możliwe jest przechwycenie i analiza działania metod w czasie rzeczywistym, bez ingerencji w kod źródłowy aplikacji.

4.5.1 Tworzenie interceptora do metody nienatywnej na przykładzie metody `ExecutorService.submit()`

Metoda nienatywna zawiera bajtkod, co umożliwia jej bezpośrednią modyfikację. Dzięki temu możliwe jest zastosowanie interceptora bezpośrednio na danej metodzie. Proces tworzenia i przygotowania do użycia takiego interceptora przebiega w następujący sposób:

1. **Stworzenie interceptora**
   Na początku tworzony jest interceptor. Jest to klasa, w której znajduje się kod wykonywany przed i po wywołaniu docelowej metody. W przypadku `execute()` i `submit()` taki interceptor wygląda następująco:

```
1   public class ExecutorExecuteSubmitAdvice {
2   @Advice.OnMethodEnter(inline = false)
3   public static void onEnter(@Advice.Origin String method,
4       @Advice.This Executor executor) {
5       // kod wykonywany przed wywołaniem submit() i execute()
6   }
7
8   @Advice.OnMethodExit(inline = false, onThrowable = Throwable.class)
9   public static void onExit() {
10      // kod wykonywany po wywołaniu submit() i execute()
11  }
12  }
```

Listing 39: Interceptor

   W powyższym przykładzie Thread Agent wykonuje kod z metody `onEnter()` przed każdym wywołaniem metod `submit()` oraz `execute()`. Analogicznie, po zakończeniu działania tych metod wykonywany jest kod z `onExit()`.

   Ustawienie parametru `inline = false` oznacza, że kod `onEnter()` i `onExit()` nie jest bezpośrednio wstrzykiwany na początku i końcu docelowej metody. Zamiast tego, generowane jest wywołanie tych metod, które znajduje się w docelowej metodzie, natomiast właściwa logika pozostaje w klasie interceptora.

2. **Stworzenie AdviceRule**
   Następnie należy powiązać interceptor z konkretną metodą i typem. Aby to zrobić, należy dodać do listy `org.threadmonitoring.advices.AdviceHandlercreateAdvices` stworzony obiekt `AdviceRule`:

```
new AdviceRule.Builder()
        .setTypeMatcher(isSubTypeOf(Executor.class))
        .setMethodMatcher(isMethod().and(named("execute").or(named("submit"))))
        .setClassName(ExecutorExecuteSubmitAdvice.class.getName())
        .build()
```

Listing 40: AdviceRule wiążący metody `execute` i `submit` z interceptorem `ExecutorServiceExecuteSubmitAdvice`

3. **Stworzenie Transformera na podstawie podanej konfiguracji**
   Następnie Thread Agent tworzy transformer w metodzie `buildAgentWithAdvicesAndSubstitutions` na podstawie przygotowanego modelu, który definiuje, jakie zmiany należy wprowadzić w bajtkodzie. Na końcu agent przeprowadza retransformację klas, dzięki czemu metody `submit()` i `execute()` zaczną być monitorowane.

4.5.2 Tworzenie interceptora do metody natywnej na przykładzie metody `Thread.sleep()`

Metoda natywna nie posiada bajtkodu, dlatego nie jest możliwe bezpośrednie wstrzyknięcie do niej wywołań metod monitorujących. Aby mimo to umożliwić jej monitorowanie, Thread Agent stosuje inną strategię — dokonuje transformacji klas, które wywołują daną metodę natywną. Zamiast bezpośredniego wywołania metody `sleep()`, agent podmienia jej wywołania na własną metodę pośredniczącą. Podmieniona metoda najpierw loguje informacje, a następnie wywołuje oryginalną metodę `sleep()`. Proces tworzenia i przygotowania do użycia takiego interceptora różni się od tego dla metod nienatywnych, ponieważ modyfikowane są nie same metody, lecz miejsca, w których są one używane.

1. **Stworzenie metody zamiennej**
   Najpierw należało stworzyć klasę i metodę, która będzie wywoływana zamiast metody `sleep`:

```java
public class SleepSubstitution {

    public static void sleep2(long millis) throws InterruptedException {
        LoggingNotifier.log(
                "Thread " + Thread.currentThread().getName()
                    + " started sleeping for " + millis + " ms"
                , SleepSubstitution.class
                , "INFO");
        Thread.sleep(millis);
        LoggingNotifier.log(
                "Thread " + Thread.currentThread().getName()
                    + " finished sleeping for " + millis + " ms"
                , SleepSubstitution.class
                , "INFO");
    }
}
```

Listing 41: Definicja metody, która będzie wykonywana zamiast `Thread.sleep()`

Zamiast wywoływać metodę `sleep()`, aplikacja użytkownika będzie wywoływać metodę `sleep2()`, przedstawioną powyżej. Metoda ta ma prostą implementację: loguje informację o swoim uruchomieniu, następnie wywołuje docelowe uśpienie wątku, a na końcu loguje informację o zakończeniu uśpienia.

Ważne jest, aby umieścić klasę zawierającą metodę `sleep2()` w projekcie `thread-agent-api`, ponieważ musi być ona dostępna z poziomu aplikacji użytkownika. Należy pamiętać, że klasy z projektu `thread-agent` nie są dostępne bezpośrednio w aplikacji użytkownika.

2. **Stworzenie MethodSubstitutionRule**
   Następnie należy powiązać metodę `sleep2()` z oryginalną metodą, którą ma zastąpić. W tym celu trzeba dodać odpowiednią regułę do listy zwracanej przez metodę `AdviceHandlercreateSubstitutions()`. W ramach tej konfiguracji należy utworzyć obiekt `MethodSubstitutionRule`, wskazując w nim, która metoda powinna zostać zastąpiona i jaka metoda ma ją zastąpić.

```
new MethodSubstitutionRule.Builder()
    .setTypeMatcher(matcher)
    .setSubstituteMethod(named("sleep")
            .and(takesArguments(long.class))
            .and(returns(void.class)))
    .setNewMethod(new MethodTemplate.Builder()
            .setClazz(SleepSubstitution.class)
            .setMethodName("sleep2")
            .setArguments(List.of(long.class))
            .build())
    .build()
```

Listing 42: MethodSubstitutionRule, który łączy natywną metodę `sleep` z podmienioną metodą `sleep2`

3. Następnie Thread Agent tworzy `Transformer` na podstawie modelu, który aplikuje zmiany bajtów. Na koniec Thread Agent przygotowuje transformację klas i w momencie załadowania klas użytkownika sprawdzi, czy te klasy wywołują metodę `sleep()`. Jeżeli tak, to podmieni ich wywołanie na metodę `sleep2()`.

4.6 Wykorzystywanie biblioteki ASM

Thread Agent wykorzystuje bibliotekę ASM do monitorowania bloków `synchronized`. W bajtkodzie bloki te reprezentowane są przez instrukcje `MONITORENTER` oraz `MONITOREXIT`, których nie da się przechwycić przy użyciu wysokopoziomowego Byte Buddy. Modyfikacja bajtkodu w tym przypadku polega na wstrzyknięciu wywołań metod logujących w trzech miejscach: przed instrukcją `MONITORENTER` (czyli w momencie oczekiwania na dostęp do bloku), tuż po jej wykonaniu (po wejściu do bloku) oraz po instrukcji `MONITOREXIT` (po opuszczeniu bloku).

```java
@Override
public void visitInsn(int opcode) {
    switch (opcode) {
        case Opcodes.MONITORENTER:
            injectLogger("logEnter");
            super.visitInsn(Opcodes.MONITORENTER);
            super.visitMethodInsn(
                    Opcodes.INVOKESTATIC,
                    "org/threadmonitoring/logger/SynchronizedLogger",
                    "logEnter2",
                    "()V",
                    false
            );
            break;
        case Opcodes.MONITOREXIT:
            injectLogger("logExit");
            super.visitInsn(Opcodes.MONITOREXIT);
            break;
        default:
            super.visitInsn(opcode);
            break;
    }
}

private void injectLogger(String methodName) {
    super.visitInsn(Opcodes.DUP);
    super.visitMethodInsn(
            Opcodes.INVOKESTATIC,
            "org/threadmonitoring/logger/SynchronizedLogger",
            methodName,
            "(Ljava/lang/Object;)V",
            false
    );
}
```

Listing 43: Kod odpowiedzialny za wywoływanie metod logujących przed i po bloku `synchronized`

W powyższym kodzie znajduje się nadpisana metoda `visitInsn`, aby przechwytywać instrukcje bajtkodu związane z synchronizacją (`MONITORENTER` i `MONITOREXIT`). W momencie wejścia do sekcji krytycznej i wyjścia z niej wstrzykiwane są dodatkowe wywołania metod loggera z klasy `SynchronizedLogger`, które rejestrują te zdarzenia. Dzięki temu możliwe jest monitorowanie i śledzenie użycia bloków synchronizacyjnych w aplikacji.

Rozdział 5

Praktyczne przykłady użycia Thread Agenta

W tym rozdziale przedstawiono praktyczne przykłady zastosowań, w których Thread Agent wspomaga użytkownika w analizie działania aplikacji. W poszczególnych podrozdziałach zaprezentowano jedynie kluczowe fragmenty kodu oraz wybrane, istotne informacje znalezione w logach.[18] Ponieważ Thread Agent generuje dużą liczbę wpisów, nie wszystkie z nich są istotne w danym czasie dla użytkownika. W praktyce konieczne jest ich filtrowanie — najczęściej na podstawie nazwy konkretnego wątku lub referencji obiektu.

5.1 Przykład edukacyjny

Thread Agent może służyć jako narzędzie wspomagające zrozumienie mechanizmów współbieżności w języku Java. W ramach tej sekcji przygotowano kod demonstracyjny, który ilustruje działanie podstawowych konstrukcji wykorzystywanych przy programowaniu wielowątkowym. Aby dobrze zobrazować działanie Thread Agenta, program został podzielony na fragmenty, w których występują różne wielowątkowe mechanizmy, które Thread Agent obrazuje.

Program został uruchomiony z Thread Agentem poprawnie, o czym świadczą poniższe logi

```
2025-09-02 15:36:43 INFO  ThreadAgent - Initializing Thread Agent before the target application to enable
↪  thread and ExecutorService monitoring
2025-09-02 15:36:43 INFO  ThreadAgent - The logging has been configured to the
↪  C:\Users\Piotr\OneDrive\Pulpit\Studia\Magisterka\threadmonitoring/logs
2025-09-02 15:36:43 INFO  ThreadAgent - Initialized advices
2025-09-02 15:36:46 INFO  ThreadAgent - Advices and method substitutions created and installed
2025-09-02 15:36:46 INFO  ThreadAgent - Attempting to retransform classes
2025-09-02 15:36:47 INFO  ThreadAgent - Retransformation completed successfully
2025-09-02 15:36:47 INFO  ThreadAgent - Transformation and Retransformation finished, Thread Agent is working,
↪  running target application...
```

Listing 44: Początkowe logi Thread Agenta

Następnie program zaczyna być monitorowany przez Thread Agenta.

```
Object lock = new Object();
int counter = 0;
Thread t1 = new Thread(() -> {
    synchronized (lock) {
        lock.wait();
    }
});
Thread t2 = new Thread(() -> {
    synchronized (lock) {
        lock.wait();
    }
});
t1.start();
t2.start();
```

Listing 45: Fragment programu `education` - dwa wątki oczekujące na wybudzenie

```
2025-09-02 15:36:48 INFO  ThreadConstructorAdvice - Thread Thread-3 created independently
2025-09-02 15:36:48 INFO  ThreadConstructorAdvice - Thread Thread-4 created independently
2025-09-02 15:36:48 INFO  ThreadStartAdvice - Started new thread - Thread[#47,Thread-3,5,main]
2025-09-02 15:36:48 INFO  ThreadStartAdvice - Started new thread - Thread[#48,Thread-4,5,main]
2025-09-02 15:36:48 INFO  SynchronizedLogger - Thread Thread-3 waiting to hold monitor:
↪  java.lang.Object@3a16442d
2025-09-02 15:36:48 INFO  SynchronizedLogger - Thread Thread-4 waiting to hold monitor:
↪  java.lang.Object@3a16442d
2025-09-02 15:36:48 INFO  SynchronizedLogger - Thread Thread-3 holds monitor: java.lang.Object@3a16442d
2025-09-02 15:36:48 INFO  WaitSubstitution - Thread Thread-3 called wait() on object: java.lang.Object@3a16442d
2025-09-02 15:36:48 INFO  SynchronizedLogger - Thread Thread-3 released monitor: java.lang.Object@3a16442d
2025-09-02 15:36:48 INFO  SynchronizedLogger - Thread Thread-4 holds monitor: java.lang.Object@3a16442d
2025-09-02 15:36:48 INFO  WaitSubstitution - Thread Thread-4 called wait() on object: java.lang.Object@3a16442d
2025-09-02 15:36:48 INFO  SynchronizedLogger - Thread Thread-4 released monitor: java.lang.Object@3a16442d
```

Listing 46: Logi Thread Agenta - wejście do bloku `synchronized` oraz oczekiwanie na wybudzenie

Wygenerowane logi wskazują, że program uruchomił dwa wątki: `Thread-3` i `Thread-4`. Oba rozpoczęły oczekiwanie na monitor, przy czym ostatecznie dostęp do niego uzyskał wątek `Thread-3`. Następnie oba wątki wywołały metodę `wait()` i oczekują na powiadomienie na obiekcie `java.lang.Object@3a16442d`.

```
Thread notifier = new Thread(() -> {
    synchronized (lock) {
        lock.notifyAll();
    }
});
notifier.start();
```

Listing 47: Fragment programu `education` - budzenie wątków

```
2025-09-02 15:36:49 INFO  ThreadConstructorAdvice - Thread Thread-5 created independently
2025-09-02 15:36:49 INFO  ThreadStartAdvice - Started new thread - Thread[#49,Thread-5,5,main]
2025-09-02 15:36:49 INFO  SleepSubstitution - Thread main started sleeping for 1000 ms
2025-09-02 15:36:49 INFO  SynchronizedLogger - Thread Thread-5 waiting to hold monitor:
↪  java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  SynchronizedLogger - Thread Thread-5 holds monitor: java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  NotifyAllSubstitution - Thread Thread-5 called notifyAll() on object:
↪  java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  SynchronizedLogger - Thread Thread-5 released monitor: java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  SynchronizedLogger - Thread Thread-3 holds monitor: java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  WaitSubstitution - Thread Thread-3 got notified on object: java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  SynchronizedLogger - Thread Thread-3 released monitor: java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  SynchronizedLogger - Thread Thread-4 holds monitor: java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  WaitSubstitution - Thread Thread-4 got notified on object: java.lang.Object@3a16442d
2025-09-02 15:36:49 INFO  SynchronizedLogger - Thread Thread-4 released monitor: java.lang.Object@3a16442d
```

Listing 48: Logi Thread Agenta - wywołanie `notifyAll()`

Wątek o nazwie `Thread-5` został uruchomiony i wywołał metodę `notifyAll()`. W efekcie wszystkie wątki oczekujące na obiekcie `java.lang.Object@3a16442d` zostały wybudzone. Gdyby zamiast tego użyto metody `notify()`, wybudzony zostałby tylko jeden, losowy wątek – informacja o tym również pojawiłaby się w logach ze wskazaniem, który wątek został wybudzony.

```
1 ExecutorService executor = Executors.newSingleThreadExecutor();
2 executor.submit(() -> System.out.println("Working..."));
3 executor.shutdown();
```

Listing 49: Fragment programu `education` - utworzenie `ExecutorService`

```
2025-09-02 15:36:50 INFO  ExecutorServiceConstructorAdvice - Thread main created new
↪  java.util.concurrent.ThreadPoolExecutor@1e96c1d7[Running, pool size = 0, active threads = 0, queued tasks =
↪  0, completed tasks = 0] at org.example.Main.main(Main.java:48)
2025-09-02 15:36:50 INFO  ExecutorServiceConstructorAdvice - Thread main created new
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@5144acb6 at
↪  org.example.Main.main(Main.java:48)
2025-09-02 15:36:50 INFO  ExecutorServiceExecuteSubmitAdvice - Task submitted by thread <main> at
↪  org.example.Main.main(Main.java:49) on
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@5144acb6
2025-09-02 15:36:50 INFO  ThreadConstructorAdvice - Thread pool-2-thread-1 created by ExecutorService:
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@5144acb6
2025-09-02 15:36:50 INFO  ThreadStartAdvice - Started new thread - Thread[#50,pool-2-thread-1,5,main]
2025-09-02 15:36:50 INFO  ExecutorServiceShutdownAdvice - ExecutorService
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@5144acb6 shutdown
2025-09-02 15:36:50 INFO  ExecutorServiceShutdownAdvice - ExecutorService
↪  java.util.concurrent.ThreadPoolExecutor@1e96c1d7[Running, pool size = 1, active threads = 0, queued tasks =
↪  0, completed tasks = 1] shutdown
```

Listing 50: Logi Thread Agenta - informacje naokoło `ExecutorService`

Logi wskazują, że program utworzył instancję `ExecutorService` w lokalizacji `org.example.Main.main(Main.java:48)`, który następnie wykonał zadanie kończąc na wywołaniu `shutdown()`. Dodatkowo zawarto informacje o tym, jakiego konkretnego typu `ExecutorService` użyła Java oraz jaki wątek i kiedy został utworzony.

```
Thread inc1 = new Thread(() -> {
    for (int i = 0; i < 2; i++) {
        synchronized (Main.class) {
            counter++;
            Thread.sleep(100);
        }
    }
});
Thread inc2 = new Thread(() -> {
    for (int i = 0; i < 2; i++) {
        synchronized (Main.class) {
            counter++;
            Thread.sleep(100);
        }
    }
});
```

Listing 51: Fragment programu `education` - inkrementacja zmiennej przez dwa wątki

```
2025-09-02 15:36:50 INFO  ThreadConstructorAdvice - Thread Thread-6 created independently
2025-09-02 15:36:50 INFO  ThreadConstructorAdvice - Thread Thread-7 created independently
2025-09-02 15:36:50 INFO  ThreadStartAdvice - Started new thread - Thread[#51,Thread-6,5,main]
2025-09-02 15:36:50 INFO  ThreadStartAdvice - Started new thread - Thread[#52,Thread-7,5,main]
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-6 waiting to hold monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-6 holds monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-7 waiting to hold monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SleepSubstitution - Thread Thread-6 started sleeping for 100 ms
2025-09-02 15:36:50 INFO  SleepSubstitution - Thread Thread-6 finished sleeping for 100 ms
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-6 released monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-6 waiting to hold monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-6 holds monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SleepSubstitution - Thread Thread-6 started sleeping for 100 ms
2025-09-02 15:36:50 INFO  SleepSubstitution - Thread Thread-6 finished sleeping for 100 ms
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-6 released monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-7 holds monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SleepSubstitution - Thread Thread-7 started sleeping for 100 ms
2025-09-02 15:36:50 INFO  SleepSubstitution - Thread Thread-7 finished sleeping for 100 ms
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-7 released monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-7 waiting to hold monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SynchronizedLogger - Thread Thread-7 holds monitor: class org.example.Main
2025-09-02 15:36:50 INFO  SleepSubstitution - Thread Thread-7 started sleeping for 100 ms
2025-09-02 15:36:51 INFO  SleepSubstitution - Thread Thread-7 finished sleeping for 100 ms
2025-09-02 15:36:51 INFO  SynchronizedLogger - Thread Thread-7 released monitor: class org.example.Main
```

Listing 52: Logi Thread Agenta - informacje naokoło bloku `synchronized`

W tej części logów można zaobserwować działanie bloku `synchronized`. Wątki oczekują na dostęp do monitora (`org.example.Main.class`), następnie uzyskują go, dokonują inkrementacji zmiennej, po czym zwalniają monitor. Interesującym przypadkiem jest sytuacja, w której mimo dłuższego oczekiwania na dostęp przez wątek `Thread-7`, zasób ponownie został przydzielony wątkowi `Thread-6`. Potwierdza to, że mechanizm `synchronized` nie zapewnia sprawiedliwości, czyli nie gwarantuje kolejności obsługi wątków według czasu zgłoszenia.

Podsumowując podany przykład, Thread Agent pomaga zrozumieć działanie wątków w Javie poprzez analizę wywołań metod.

5.2 Problem wycieku wątków

Wyciek wątków to sytuacja, w której wątki są tworzone, lecz nie zostają prawidłowo zakończone, co skutkuje stopniowym obciążeniem systemu i może prowadzić do wyczerpania zasobów – zarówno pamięci, jak i limitu dostępnych wątków. Do tego typu problemów najczęściej dochodzi podczas korzystania z `ExecutorService`, zwłaszcza w przypadkach, gdy zadania nigdy się nie kończą lub nie są odpowiednio zarządzane. Dodatkowo warto mieć na uwadze, że nieobsłużone wyjątki występujące w zadaniach przekazanych do `ExecutorService` nie są automatycznie propagowane na zewnątrz. Mogą one pozostać niezauważone. Choć sam wyciek wątków jest relatywnie łatwy do wykrycia – na przykład przez monitorowanie pamięci systemowej lub liczby aktywnych wątków (przykładowo przy użyciu zrzutu stanu wątków) to znacznie trudniejsze jest ustalenie jego przyczyny. Gdy źródłem problemu jest `ExecutorService`, tradycyjny zrzut stanu wątków może okazać się niewystarczający, ponieważ wątki tworzone przez `ExecutorService` są anonimowe, a ich ścieżka wywołań często ogranicza się do kodu wewnętrznego biblioteki Javy pomijając ścieżkę wywołań samego `ExecutorService`. W takich przypadkach Thread Agent jest bardzo przydatny, ponieważ rejestruje nie tylko moment uruchomienia wątków, ale również ich pochodzenie, w tym wywołania związane z `ExecutorService`. Dzięki temu możliwe jest szybkie wskazanie źródła wycieku.

```
1 while (true) {
2     ExecutorService executor = Executors.newSingleThreadExecutor();
3     executor.submit(() -> {
4         Thread.sleep(Long.MAX_VALUE);
5     });
6     Thread.sleep(3000);
7 }
```

Listing 53: Fragment programu `thread-leak` - wyciek wątków

W powyższym przykładzie, co 3 sekundy tworzony jest nowy `ExecutorService`, który uruchamia zadanie, którego zakończenie wymagałoby setek milionów lat. Taka apliakcja prowadzi do stałego wzrostu liczby aktywnych wątków, czyli wycieku wątków. Po uruchomieniu aplikacji z Thread Agentem, generowany jest plik z logami zawierający następujące informacje:

```
2025-09-02 17:41:53 INFO  ExecutorServiceConstructorAdvice - Thread main created new
↪  java.util.concurrent.ThreadPoolExecutor@1221a708[Running, pool size = 0, active threads = 0, queued tasks =
↪  0, completed tasks = 0] at org.example.Main.main(Main.java:10)
2025-09-02 17:41:53 INFO  ExecutorServiceConstructorAdvice - Thread main created new
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@1fc3db01 at
↪  org.example.Main.main(Main.java:10)
2025-09-02 17:41:53 INFO  ExecutorServiceExecuteSubmitAdvice - Task submitted by thread <main> at
↪  org.example.Main.main(Main.java:12) on
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@1fc3db01
2025-09-02 17:41:53 INFO  ThreadConstructorAdvice - Thread pool-2-thread-1 created by ExecutorService:
↪  java.util.concurrent.Executors$AutoShutdownDelegatedExecutorService@1fc3db01
2025-09-02 17:41:53 INFO  ThreadStartAdvice - Started new thread - Thread[#47,pool-2-thread-1,5,main]
2025-09-02 17:41:53 INFO  SleepSubstitution - Thread pool-2-thread-1 started sleeping for 9223372036854775807
↪  ms
2025-09-02 17:41:53 INFO  SleepSubstitution - Thread main started sleeping for 3000 ms
```

Listing 54: Logi Thread Agenta - przyczyna wycieku wątków

Powyższy fragment logów pojawia się wielokrotnie w pełnym pliku wygenerowanym przez Thread Agenta. Z pliku można wyczytać:

- Zostaje stworzony `ExecutorService`.
- Obiekt `ExecutorService` natychmiast uruchamia zadanie, w którym wątek zostaje uśpiony na bardzo długi czas.
- Brak informacji w logach o zamknięciu ExecutorService.
- Nowy `ExecutorService` tworzony jest regularnie w `org.example.Main.main(Main.java:10)`.

Dzięki logom można łatwo zidentyfikować bezpośrednią przyczynę problemu: powtarzające się tworzenie nowych `ExecutorService` w `org.example.Main.main(Main.java:10)` bez ich zamykania oraz uruchamianie zadań, które nie mają szans się zakończyć. Dodatkowym potwierdzeniem problemu są niezakończone wątki podczas zamknięcia programu widoczne po przerwaniu działania aplikacji w `thread-agent1_emergency`:

```
[2025-09-02 17:42:02] Thread pool-3-thread-1 was forced to terminate during JVM shutdown with state:
↪  TIMED_WAITING
Stacktrace:
        at java.base/java.lang.Thread.sleep0(Native Method)
        at java.base/java.lang.Thread.sleep(Thread.java:509)
        at org.threadmonitoring.substitution.SleepSubstitution.sleep2(SleepSubstitution.java:12)
        at org.example.Main.lambda$main$0(Main.java:14)
        at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:572)
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:642)
        at java.base/java.lang.Thread.run(Thread.java:1583)

[2025-09-02 17:42:02] Thread pool-4-thread-1 was forced to terminate during JVM shutdown with state:
↪  TIMED_WAITING
Stacktrace:
        at java.base/java.lang.Thread.sleep0(Native Method)
        at java.base/java.lang.Thread.sleep(Thread.java:509)
        at org.threadmonitoring.substitution.SleepSubstitution.sleep2(SleepSubstitution.java:12)
        at org.example.Main.lambda$main$0(Main.java:14)
        at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:572)
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:642)
        at java.base/java.lang.Thread.run(Thread.java:1583)
```

Listing 55: Logi Thread Agenta - niezakończone wątki

Jest to prosty przykład ilustrujący wykrywanie wycieku wątków, który można łatwo zauważyć bez używania dodatkowych zewnętrznych narzędzi. W przypadku bardziej złożonych systemów kluczowe staje się precyzyjne zlokalizowanie źródła problemu, zwłaszcza gdy aplikacja zarządza dziesiątkami, a nawet setkami instancji klasy `ExecutorService`.

5.3 Problem synchronizacji wątków

Na potrzeby demonstracji działania Thread Agenta oraz identyfikacji problemów synchronizacji wątków wykorzystano przykład stosowany podczas zajęć "Programowanie rozproszone i równoległe"[19]. Fragment poniższego kodu został udostępniony przez dra Piotra Oramusa na potrzeby niniejszej pracy. Przedstawiona aplikacja symuluje przekazywanie piłki między wątkami reprezentującymi graczy.

```
class Player implements Runnable {
    public void run() {
        while ( true ) {
            synchronized (ball) {
                if (ball.getDestination() == myNumber) {
                    destination = random.getOtherPlayerNumber(myNumber);
                    ball.setDestination(destination);
                    ball.notify();
                } else {
                    ball.wait();
                }
            }
        }
    }
}

class Start {
    private static final int NUMBER_OF_PLAYERS = 22;

    public static void main( String[] argv ) {

        Ball ball = new Ball();
        Random rnd = new Random( NUMBER_OF_PLAYERS );

        for ( int i = 0; i < NUMBER_OF_PLAYERS; i++ ) {
            ( new Thread( new Player( i, ball, rnd ) ) ).start();
        }
    }
}
```

Listing 56: Fragmenty programu `pilka`

Kod implementuje model, w którym wątek reprezentujący gracza przekazuje piłkę innemu graczowi. Wątek wchodzi do sekcji krytycznej, jeżeli jest odbiorcą piłki to przekazuję ją dalej, wywołuje `notify()` na współdzielonym obiekcie (piłce), a następnie przechodzi w stan oczekiwania, wywołując `wait()`. Kolejny wątek powinien zostać wybudzony i kontynuować przekazywanie piłki dalej. Patrząc na kod, wszystko wydaje się poprawne. Jednak podczas działania aplikacji w środowisku wielowątkowym pojawia się problem.

```
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-7 called wait() on object: org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-10 called wait() on object:
↪  org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-6 called wait() on object: org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-11 called wait() on object:
↪  org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-3 called wait() on object: org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-12 got notified on object: org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-12 called wait() on object:
↪  org.example.Ball@6de5c3ec
```

Listing 57: Logi Thread Agenta wskazujące na obudzenie wątku, który nie jest odbiorcą piłki

Thread Agent generuje logi, które jasno wskazują na problem. Wszystkie wątki próbują oczekiwać na tym samym obiekcie – `org.example.Ball@6de5c3ec`. W rezultacie wszystkie wątki zostają zablokowane, czekając na `notify()`, który mimo wywołania, nie daje gwarancji wybudzenia właściwego odbiorcy. W logach można zauważyć, że obudzony został wątek `Thread-12`, który nie był odbiorcą piłki, więc nie mógł kontynuować działania i ponownie przeszedł w stan oczekiwania. Zastosowanie `notify()` skutkuje wybudzeniem dowolnego, jednego wątku oczekującego na danym obiekcie – bez gwarancji, że będzie to, jak w przypadku tego programu, właściwy odbiorca.

Aby zwiększyć przejrzystość logów, można rozważyć dodanie dodatkowych informacji, takich jak identyfikatory graczy biorących udział w przekazaniu piłki (np. który gracz przekazuje piłkę do którego). Choć nie jest to wymagane — ponieważ aplikacja wypisuje te dane na standardowe wyjście, a sam Thread Agent potrafi wskazać problem w logach bez modyfikacji kodu — to taka zmiana może znacząco ułatwić analizę. Dzięki temu kluczowe informacje będą dostępne bezpośrednio w logach, tuż obok pozostałych danych związanych z wywołaniami metod. Aby tego dokonać, należy dopisać fragment kodu - `Call.alertMultithreadingCall("setDestination("+ destination + ")");`:

```
1 if (ball.getDestination() == myNumber) {
2     destination = random.getOtherPlayerNumber(myNumber);
3     ball.setDestination(destination);
4     Call.alertMultithreadingCall("setDestination(" + destination + ")");
5     ball.notify();
6 }
```

Listing 58: Zmodyfikowany fragment programu `pilka` - dodanie logowania odbiorcy piłki

Po dokonaniu zmiany w kodzie, Thread Agent potwierdza, że odbiorcą jest inny użytkownik:

```
2025-09-02 18:02:53 INFO  GeneralSubstitution - Thread Thread-3 called setDestination(2)
2025-09-02 18:02:53 INFO  NotifySubstitution - Thread Thread-3 called notify() on object:
↪  org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-12 got notified on object: org.example.Ball@6de5c3ec
2025-09-02 18:02:53 INFO  WaitSubstitution - Thread Thread-12 called wait() on object:
↪  org.example.Ball@6de5c3ec
```

Listing 59: Logi Thread Agenta po dodaniu logowania odbiorcy piłki

Dzięki dodanym informacjom możliwe jest jednoznaczne zidentyfikowanie, że `Thread-3` wywołał `notify()` z zamiarem obudzenia gracza `2`, jednak obudzony został `Thread-12`, który nie był odbiorcą i natychmiast ponownie wywołał `wait()`. Taki problem można rozwiązać na dwa sposoby:

- **Zamiana `notify()` na `notifyAll()`**
  Umożliwia wybudzenie wszystkich wątków oczekujących na obiekcie. Tylko wątek spełniający warunek logiczny kontynuuje działanie, pozostałe wracają do oczekiwania. Rozwiązanie eliminuje problem przypadkowego budzenia niewłaściwego wątku.

- **`Lock` z wieloma obiektami `Condition`**
  Każdy zawodnik oczekuje na własnym obiekcie typu `Condition`, gdy piłka nie jest do niego skierowana. Po zmianie adresata piłki wywoływana jest metoda `signal()` na obiekcie `Condition` przypisanym do właściwego gracza, co powoduje wybudzenie tylko odpowiedniego wątku.

Przykład ten pokazuje, w jaki sposób Thread Agent umożliwia nieinwazyjne logowanie istotnych informacji, które pozwalają szybko zidentyfikować zarówno skutki, jak i przyczyny problemów z synchronizacją w aplikacjach wielowątkowych. Dodatkowo, dzięki integracji z biblioteką Monitoring Call, możliwe jest wzbogacenie logów o dodatkowy kontekst, co pozwala na dodatkowe ułatwienie analizy działania aplikacji z perspektywy użytkownika.

5.4 Problem zakleszczenia wątków

Thread Agent umożliwia nie tylko wykrycie wystąpienia zakleszczenia w aplikacji wielowątkowej, ale również identyfikację jego bezpośredniej przyczyny. Narzędzie analizuje operacje wykonywane na obiektach synchronizujących, takich jak `Lock`, rejestrując, które wątki próbują uzyskać dostęp do zasobów oraz które z nich aktualnie je blokują. Poniżej przedstawiono przykład kodu, który prowadzi do takiej sytuacji.

```
1  Lock A = new ReentrantLock();
2  Lock B = new ReentrantLock();
3  new Thread(() -> {
4      A.lock();
5      Thread.sleep(50);
6      B.lock();
7  }).start();
8  new Thread(() -> {
9      B.lock();
10     Thread.sleep(50);
11     A.lock();
12 }).start();
```

Listing 60: Przykład zakleszczenia w Javie

Po uruchomieniu aplikacji wraz z Thread Agentem, w logach można zauważyć informacje dotyczące tworzenia wątków oraz prób wywołania metody `lock()`. Początkowe wywołania `lock()` kończą się sukcesem, ponieważ obiekty klasy `Lock` mają początkowo status `Unlocked`.

```
2025-09-02 18:25:37 INFO  ThreadConstructorAdvice - Thread Thread-3 created independently
2025-09-02 18:25:37 INFO  LockAdvice - Lock java.util.concurrent.locks.ReentrantLock@b0e0648[Locked by thread
↪  Thread-3] acquired by thread Thread-3 at org.example.Main.lambda$main$0(Main.java:13)
2025-09-02 18:25:37 INFO  ThreadConstructorAdvice - Thread Thread-4 created independently
2025-09-02 18:25:37 INFO  LockAdvice - Lock java.util.concurrent.locks.ReentrantLock@1b8f68c7[Locked by thread
↪  Thread-4] acquired by thread Thread-4 at org.example.Main.lambda$main$1(Main.java:22)
```

Listing 61: Logi Thread Agenta wskazujące na przejęcie blokady przez oba wątki

Następnie oba wątki próbują zablokować obiekty `Lock`, które są już zajęte przez siebie nawzajem:

```
2025-09-02 18:25:37 INFO  LockAdvice - Lock java.util.concurrent.locks.ReentrantLock@1b8f68c7[Locked by thread
↪  Thread-4] waiting to be acquired by thread Thread-3 at org.example.Main.lambda$main$0(Main.java:19)
2025-09-02 18:25:37 INFO  LockAdvice - Lock java.util.concurrent.locks.ReentrantLock@b0e0648[Locked by thread
↪  Thread-3] waiting to be acquired by thread Thread-4 at org.example.Main.lambda$main$1(Main.java:28)
```

Listing 62: Logi Thread Agenta wskazujące na próbę przejęcia blokad przez oba wątki

W logach brakuje informacji o zwolnieniu zasobów przez którykolwiek z wątków, co wskazuje na potencjalne wystąpienie zakleszczenia pomiędzy `Thread-3` i `Thread-4`. Dodatkowo w logu pojawiła się informacja o wykryciu potencjalnego zakleszczenia.

```
2025-09-02 18:25:37 ERROR DeadlockAnalyzer - Potential deadlock detected! Please check emergency log for
↪  details
```

Listing 63: Logi Thread Agenta wskazujące na wykrycie potencjalnego zakleszczenia

```
[2025-09-02 18:25:37] Potential deadlock detected!
[2025-09-02 18:25:37] Deadlock cycle:
[2025-09-02 18:25:37] Thread[#48,Thread-4,5,main] -> Thread[#47,Thread-3,5,main] -> Thread[#48,Thread-4,5,main]
[2025-09-02 18:25:37] Stack traces of involved threads:
[2025-09-02 18:25:37] Thread: Thread-4 (ID: 48)
[2025-09-02 18:25:37]     at org.example.Main.lambda$main$1(Main.java:28)
[2025-09-02 18:25:37]
[2025-09-02 18:25:37] Thread: Thread-3 (ID: 47)
[2025-09-02 18:25:37]     at java.base/jdk.internal.misc.Unsafe.park(Native Method)
[2025-09-02 18:25:37]     at org.example.Main.lambda$main$0(Main.java:19)
```

Listing 64: Zawartość `thread-agent1_emergency.log` - potencjalne zakleszczenie i ścieżki wywołań wątków uczestniczących

Podsumowując, Thread Agent prawidłowo wykrył zakleszczenie, wynikające z wzajemnego oczekiwania wątków na zasoby zablokowane przez siebie nawzajem.

5.5 Wywołanie `wait/notify` bez uprzedniego przejęcia monitora

Thread Agent analizuje każde wywołanie metod `wait` oraz `notify`. Jeżeli wątek próbuje je wywołać bez posiadania monitora danego obiektu, aplikacja zgłasza wyjątek `IllegalMonitorStateException`. Thread Agent dodatkowo przechwytuje taką sytuację i zapisuje odpowiedni komunikat w pliku `emergency`. Poniżej przedstawiono przykład kodu prowadzącego do wystąpienia tego wyjątku:

```
Thread.ofPlatform().start(new Runnable() {
    public void run() {
        Main.class.wait(1000);
    }
});
Thread.ofPlatform().start(new Runnable() {
                                public void run() {
                                    Main.class.notify();
                                }
                            }
);
```

Listing 65: Przykład błędnego wywołania metody `wait()` i `notify()`

Thread Agent wygenerował następujące logi:

```
2025-09-02 20:25:42 INFO  ThreadConstructorAdvice - Thread Thread-3 created independently
2025-09-02 20:25:42 INFO  ThreadStartAdvice - Started new thread - Thread[#47,Thread-3,5,main]
2025-09-02 20:25:42 INFO  ThreadConstructorAdvice - Thread Thread-4 created independently
2025-09-02 20:25:42 INFO  ThreadStartAdvice - Started new thread - Thread[#48,Thread-4,5,main]
2025-09-02 20:25:42 INFO  WaitSubstitution - Thread Thread-3 called wait(1000) on object: class
↪  org.example.Main
2025-09-02 20:25:42 INFO  NotifySubstitution - Thread Thread-4 called notify() on object: class
↪  org.example.Main
```

Listing 66: Logi Thread Agenta wskazujące na błędne wywołanie metody `wait(1000)` i `notify()`

W logach nie występuje ani przechwycenie bloku `synchronized`, ani wywołanie metody oznaczonej jako `synchronized`, co wskazuje na brak uprzedniego przejęcia monitora. Na tej podstawie można podejrzewać problem i spodziewać się wystąpienia wyjątku, co znajduje potwierdzenie w pliku `emergency`.

```
[2025-09-02 20:25:42] Thread Thread-3 called wait(1000) on object: class org.example.Main without holding the
↪  monitor on required object! Application will throw IllegalMonitorStateException
[2025-09-02 20:25:42] Thread Thread-4 called notify() on object: class org.example.Main without holding the
↪  monitor on required object! Application will throw IllegalMonitorStateException
```

Listing 67: Logi z pliku `emergency` wskazujące na wywołanie metod `wait(1000)` oraz `notify()` bez uprzedniego przejęcia monitora obiektu

W tych logach błąd jest wyraźnie widoczny — wątki wywołują `wait` i `notify` bez uprzedniego przejęcia monitora obiektu.

Rozdział 6

Plany na rozwój aplikacji

Thread Agent może być rozwijany w wielu kierunkach, aby zwiększyć jego funkcjonalność, użyteczność oraz możliwości analizy problemów związanych z wielowątkowością. Poniżej przedstawiono główne obszary rozwoju.

6.1 Graficzny interfejs

Obecnie aplikacja loguje zdarzenia do plików, a użytkownik musi samodzielnie analizować dane, aby wyciągnąć z nich istotne informacje. W przyszłości można rozszerzyć możliwości Thread Agenta o eksport danych w formacie JSON, a następnie przesyłać je do serwera.

Na serwerze dane mogą zostać przetworzone i zaprezentowane w przejrzystym, graficznym interfejsie użytkownika, przykładowo w formie wykresów czasowych czy tabel. Taki interfejs mógłby zostać zrealizowany przy użyciu architektury opartej na Apache Kafka[20] do przesyłania danych, Spring Boot[21] jako backend serwera oraz React[22] do budowy frontendu.

Dodatkowo, interfejs graficzny umożliwiłby zdalny dostęp do danych bez konieczności logowania się na serwer i ręcznego przeglądania logów. Taki system mógłby również umożliwiać wykonywanie akcji na żywo — przykładowo zdalne wywołanie zrzutu stanu wątków za pomocą przycisku w interfejsie.

6.2 Dodanie wsparcia dla dodatkowych metod

Choć Thread Agent już teraz monitoruje wiele istotnych metod związanych z synchronizacją wątków, istnieje możliwość jego dalszego rozszerzenia o dodatkowe mechanizmy współbieżności dostępne w Javie. Umożliwi to dokładniejszą diagnostykę i analizę bardziej złożonych problemów wielowątkowych.

Poniżej przedstawiono listę metod i klas, które warto dodać do monitorowania:

- `Semaphore` – klasa z pakietu `java.util.concurrent`, pozwala na ograniczenie liczby jednoczesnych dostępów do zasobu. Śledzenie jej metod (np. `acquire()` i `release()`) daje obraz, jak zasoby są współdzielone między wątkami.
- `CyclicBarrier` – synchronizator pozwalający grupie wątków poczekać na siebie wzajemnie w określonym punkcie programu. Monitorowanie momentów, w których wątki

dochodzą do bariery, może ujawnić problemy związane z niespełnionymi warunkami synchronizacji.

- `CountDownLatch` – umożliwia jednemu lub wielu wątkom oczekiwanie, aż inne wątki wykonają zestaw operacji. Rejestrowanie użycia `await()` i `countDown()` pozwala lepiej zrozumieć zależności między różnymi etapami wykonywania programu.
- `BlockingQueue` – interfejs kolejek blokujących, które są szczególnie przydatne w architekturach producent-konsument. Monitorowanie operacji takich jak `put()` i `take()` pozwala analizować przepływ danych między wątkami.

Thread Agent został zaprojektowany w sposób umożliwiający łatwe rozszerzanie o nowe metody do monitorowania. Dzięki przygotowanemu modelowi dodanie wsparcia dla kolejnych metod nie wymaga zaawansowanej znajomości biblioteki Byte Buddy — wystarczy zastosować istniejący szablon opisany w rozdziale 4.5.

6.3 Dodanie dodatkowych funkcji

Thread Agent może zostać wzbogacony o nowe funkcje, które ułatwią programistom analizę działania aplikacji wielowątkowych i diagnozowanie problemów związanych z współbieżnością. Poniżej przedstawiono propozycje rozszerzeń wykraczających poza samo monitorowanie metod.

- **Zrzuty stanu wątków w regularnych odstępach czasu**

  Automatyczne wykonywanie zrzutów stanu wątków w określonych interwałach czasowych umożliwia obserwację dynamiki pracy wątków w czasie. Dzięki temu możliwe jest śledzenie zmian w ich stanie i lokalizowanie potencjalnych źródeł problemów.

- **Cykliczne skanowanie stanu wątków**

  Po uruchomieniu Thread Agenta można uruchomić dedykowany wątek, który będzie w sposób ciągły monitorował i logował stan wszystkich aktywnych wątków w systemie w określonych przedziałach czasowych. Pozwoli to tworzyć bardziej dokładny obraz pracy aplikacji w czasie rzeczywistym.

- **Rozszerzona konfiguracja**

  Obecnie Thread Agent pozwala na konfigurację monitorowania wyłącznie na poziomie pakietów. Warto rozszerzyć tę funkcjonalność o możliwość wyłączania monitorowania poszczególnych metod, które z perspektywy użytkownika nie mają istotnego znaczenia.

- **Elastyczny format logowania**

  Aktualnie dane są logowane przy użyciu biblioteki `log4j`. Przyszłe wersje mogłyby umożliwiać zapis danych także w formacie JSON lub jako czysty tekst, co ułatwi ich dalsze przetwarzanie i integrację z narzędziami analitycznymi lub systemami monitorowania.

Rozdział 7

Podsumowanie

7.1 Podsumowanie pracy

Thread Agent został zaprojektowany jako narzędzie kompatybilne z każdym programem Java od wersji 11. Jego głównym celem jest szczegółowa analiza działania wątków w czasie rzeczywistym, co umożliwia głębsze zrozumienie mechanizmów współbieżności w Javie oraz identyfikację typowych problemów wielowątkowych.

Aplikacja modyfikuje kod zarówno aplikacji użytkownika, jak i wybranych klas systemowych Javy, co pozwala wykrywać potencjalne błędy synchronizacji, wycieki wątków oraz inne trudne do zdiagnozowania problemy związane z równoległym przetwarzaniem. Dzięki temu Thread Agent może służyć nie tylko jako narzędzie diagnostyczne, ale również jako zaawansowana pomoc edukacyjna, prezentująca niskopoziomowe aspekty działania klas standardowej biblioteki Javy.

W pracy podkreślono, że współbieżność pozostaje jednym z największych wyzwań w programowaniu w Javie, a istniejące narzędzia diagnostyczne często koncentrują się na skutkach problemów, a nie ich źródłach. Thread Agent, jako rozwiązanie oparte na agencie Java, prezentuje alternatywne podejście, pozwala analizować procesy prowadzące do problemów, oferując przy tym dużą elastyczność i minimalne wymagania konfiguracyjne ze strony użytkownika.

7.2 Doświadczenie

Podczas realizacji projektu pojawiło się wiele złożonych wyzwań technicznych, głównie związanych z niskopoziomowymi aspektami języka Java i jego środowiska wykonawczego. Implementacja Thread Agenta pozwoliła znacząco pogłębić wiedzę z zakresu zaawansowanych tematów, takich jak: mechanizmy działania ClassLoaderów, instrumentacja klas, analiza bajtkodu, interceptory, agenci Java, czy analiza klas JDK.

Zrozumienie i praktyczne zastosowanie tych zagadnień pozwoliło nie tylko na stworzenie funkcjonalnego narzędzia, ale również na zdobycie cennego doświadczenia, które ma bezpośrednie przełożenie na jakość pracy z systemami wielowątkowymi w rzeczywistych projektach. Tematy te, mimo że często pomijane w codziennym programowaniu, są kluczowe dla tworzenia niezawodnych, skalowalnych i bezpiecznych aplikacji w środowisku Java.

Spis rysunków, tabel i listingów

Spis rysunków

Spis tabel

Spis listingów

Bibliografia

[1] Program JDK Mission Control

https://www.oracle.com/java/technologies/jdk-mission-control.html (dostęp: 17.06.2025)

[2] Oracle

https://www.oracle.com/ (dostęp: 17.06.2025)

[3] Java Flight Recorder to narzędzie do profilowania i monitorowania działania aplikacji Java, które rejestruje dane diagnostyczne o działaniu JVM i aplikacji

https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm (dostęp: 17.06.2025)

[4] Problemy wynikające z braku lub błędnej synchronizacji zostały szczegołówo opisane w książce Goetz B. i in., Java Concurrency in Practice. Boston: Addison-Wesley, 2006. Rozdziały 2, 3 i 10.

[5] Zakleszczenie

https://en.wikipedia.org/wiki/Deadlock_(computer_science)

(dostęp: 17.06.2025)

[6] Livelock

https://en.wikipedia.org/wiki/Deadlock_(computer_science)#Livelock

(dostęp: 17.06.2025)

[7] Zagłodzenie

https://en.wikipedia.org/wiki/Starvation_(computer_science)

(dostęp: 17.06.2025)

[8] Warunek wyścigu

https://en.wikipedia.org/wiki/Race_condition

(dostęp: 17.06.2025)

[9] ASM

https://asm.ow2.io/ (dostęp: 17.06.2025)

[10] Byte Buddy

`https://bytebuddy.net/` (dostęp: 17.06.2025)

[11] Javassist

`https://www.javassist.org/` (dostęp: 17.06.2025)

[12] log4j

`https://logging.apache.org/` (dostęp: 17.06.2025)

[13] Jackson

`https://github.com/FasterXML/jackson` (dostęp: 17.06.2025)

[14] CodeCouple. Domyślne ClassLoadery w Javie. 18.01.2019

Dostępny w Internecie: `https://codecouple.pl/2019/01/18/class-loader-w-javie/` (dostęp: 17.06.2025)

[15] Wszystkie projekty znajdują się w folderze `ThreadAgent` załączonym wraz z pracą.

[16] Zbudowany projekt ze wszystkim zależnościami znajduje się w folderze `threadmonitoring` załączonym wraz z pracą.

[17] Rozwiązanie zostało zainspirowane implementacją firmy Elastic, dostępną w ramach projektu *apm-agent-java* `https://www.elastic.co` w module `apm-agent-core`, którego implementację można znaleźć pod adresem: `https://github.com/elastic/apm-agent-java/blob/main/apm-agent-core/src/main/java/co/elastic/apm/agent/bci/IndyBootstrap.java#L77`. (dostęp: 17.06.2025)

[18] Pełne wersje kodów źródłowych oraz kompletne logi są załączone wraz z pracą.

[19] Zajęcia "Programowanie rozproszone i równoległe" prowadzone przez dra Piotra Oramusa na Uniwersytecie Jagiellońskim w Krakowie.

[20] Apache Kafka

`https://kafka.apache.org/` (dostęp: 17.06.2025)

[21] Spring

`https://spring.io/` (dostęp: 17.06.2025)

[22] React

`https://react.dev/` (dostęp: 17.06.2025)