Zeszyty Naukowe
Wyższej Szkoły Ekonomii i Informatyki
Nr 20/2024, s. 175-190
DOI: XXXXXX

ISSN 1734-5391
e-ISSN 2957-2223

Janusz Rybarski
Dr inż., Zakład Informatyki,
Wyższa Szkoła Ekonomii i Informatyki
w Krakowei (WSEI)
email: jrybarski@wsei.edu.pl
ORCID: 0009-0009-1999-7983

NISKOKODOWY FRAMEWORK OPARTY NA ZDARZENIACH NAPISANY W JĘZYKU JAVA

LOW CODE EVENT BASED JAVA FRAMEWORK

Słowa kluczowe: *java, proces sterowany zdarzenia, low code framework*
Key words: *java, event-driven, framework low-code*

JEL Classification: A2, C8, I2

Streszczenie

Artykuł omawia platformę niskokodową (ang. low-code) opartą na zdarzeniach, zaimplementowaną w języku Java. Omówione zostały główne założenia systemu opartego na blokach logicznych oraz szczegóły implementacyjne obejmujące strukturę klas, mechanizm annotacji dla sygnałów wejściowych i wyjściowych oraz implementacja przykładowego bloku logicznego (koniunkcja). Na zakończenie przedstawiono przykład zastosowania rozwiązania, w którym przepływ danych oraz zachowanie aplikacji

zdefiniowane jest w pliku konfiguracyjnym XML. Przeanalizowano także potencjalne problemy związane z testowaniem systemu, wynikające głównie z jego asynchronicznej natury

Abstract
The article discusses an event-based low-code platform implemented in Java. The main assumptions of the system based on logical blocks are discussed, and the implementation details include the class structure, the annotation mechanism for input and output signals and the implementation of an example logical block (conjunction). Finally, an example of the application of the described solution is presented, in which the data flow and application behavior are defined in an XML configuration file. Potential problems related to testing the system, resulting mainly from its asynchronous nature, were also analyzed.

POJĘCIA PODSTAWOWE

Low-code – platforma niskokodowa (ang. *low-code development platform, LCDP*) – jest oprogramowaniem umożliwiającym budowę aplikacji w sposób wizualny, za pomocą diagramów, grafów czy formularzy z ograniczoną znajomością języków programowania [1]

Event-driven architecture – Architektura oparta na zdarzeniach – to styl projektowania aplikacji, w którym aplikacja reaguje na różne zdarzenia, takie jak np. naciśnięcie przycisku, przesłanie danych przez sieć czy też zmiana stanu aplikacji. W tej architekturze aplikacja składa się z wielu niezależnych komponentów, które są ze sobą połączone przez system zdarzeń. [2]

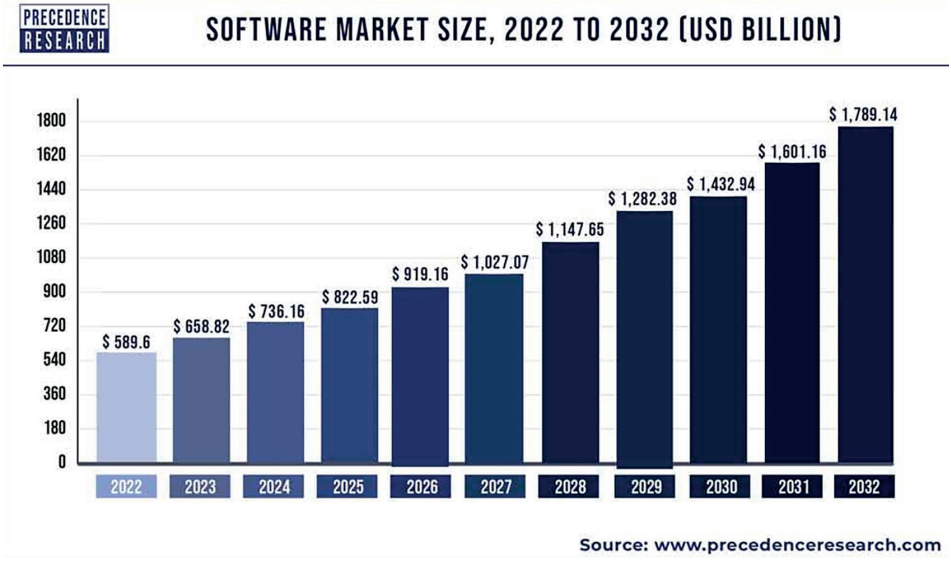
Wielowątkowość (ang. *multithreading*) – cecha systemu operacyjnego, dzięki której w ramach jednego procesu może być wykonywanych kilka zadań nazywanych wątkami. Nowe zadania to kolejne ciągi instrukcji realizowane do pewnego stopnia niezależnie. Wszystkie wątki (zadania) w ramach tego samego procesu współdzielą tę samą wirtualną przestrzeń adresową zawierającą kod programu i jego dane. [3]

Framework – szkielet do budowy aplikacji. Definiuje on strukturę aplikacji oraz ogólny mechanizm jej działania, a także dostarcza zestaw komponentów i bibliotek ogólnego przeznaczenia do wykonywania określonych zadań. [4]

WSTĘP

W ciągu ostatnich kilkunastu lat obserwujemy dynamiczny rozwój rynku IT oraz nowoczesnych technologii z nim związanych. Obecnie trudno sobie wyobrazić, abyśmy mogli funkcjonować bez udziału telefonu, komputera, Internetu, które są obecne wszędzie, począwszy od samochodów, pralek, komputerów a na zwykłych zegarkach skończywszy. Z narzędzi informatycznych korzystamy w domu, szkole, w czasie podróży, a także – co jest wyraźnie zauważalnym negatywnym trendem – coraz częściej cyfrowa rzeczywistość towarzyszy nam także w czasie wolnym. Jednocześnie, co naturalne, wraz ze wzrostem zapotrzebowania na oprogramowanie, wzrosło także zapotrzebowanie na wysoko wyspecjalizowanych pracowników sektora nowoczesnych technologii [5]. Poniżej przedstawiono prognozowaną wielkość rynku IT do 2032 roku [6] w ujęciu globalnym. Widać wyraźnie, iż szacuje się, że rynek ten zwiększy się ponad trzykrotnie.

Rysunek 1. Wielkość rynku IT 2022-2032 w MLD USD



Źródło: https://www.precedenceresearch.com/software-market (2024-03-20).

Pomimo tego, że większość szkół wyższych kształci studentów w nowoczesnych technologiach podaż pracowników w dalszym ciągu nie zaspokaja popytu, a związana z tym rosnąca dysproporcja pomiędzy podażą a popytem, powoduje znaczną presję płacową na rynku IT, oraz coraz częstsze wykluczenie małych i średnich firm w przysłowiowym „wyścigu zbrojeń".

W związku z tym, firmy chętniej, niż było to zauważalne dawniej, spoglądają w kierunku rozwiązań, które pozwalają zastąpić niedobór pracowników sektora IT, bądź też powodujących zmniejszenie zapotrzebowania na tego typu specjalistów. Z pomocą przychodzą rozwiązania oparte bądź to o generatywną sztuczną inteligencję, powszechnie nazywane rozwiązaniami AI (ang. Artificial Intelligence) lub ML (ang. Machine Learning) albo też platformy low-code. Wymienione rozwiązania, o ile nie zastąpią całkowicie pracowników sektora IT, pozwalają firmom przenieść część prac na barki sztucznej inteligencji lub osoby, których wykształcenie odbiega od wykształcenia związanego z nowoczesnymi technologiami. W szczególności platformy low-code pozwalają przenieść część prac związanych na przykład z rozwojem oprogramowania, tworzeniem automatycznego obiegu dokumentów lub innych zadań dotychczas wykonywanych przez specjalistów IT, na osoby o odmiennym profilu edukacyjnym. Platformy takie wymagają jedynie podstawowej umiejętności programowania (w przeciwieństwie do platform no-code, które nie wymagają znajomości języków programowania), a tworzone rozwiązania najczęściej budowane są w oparciu o łatwy do opanowania interfejs graficzny. Według Allied Market Research [7] globalny rynek platform low-code wzrośnie z 11,5 mld dolarów w 2021 roku do 125 mld dolarów w roku 2030. Szacuje się, iż oprogramowanie low-code będzie odgrywać znaczącą rolę w branżach takich jak [8]:

- IoT – Internet of things, gdzie skupiać się będzie głównie na zbieraniu, analizie i wizualizacji danych,
- w branżach wysoko regulowanych takich jak finanse i opieka zdrowotna,
- jako narzędzie DevOps,
- do analizy i wizualizacji danych,
- do budowania i prototypowania aplikacji mobilnych i webowych.

Niniejszy artykuł opisuje, stworzony w języku java, sterowany zdarzeniami low-code framework, który z powodzeniem wykorzystywany jest w stworzonym przez Autora systemie inteligentnego domu (ang. Smart

House). Autor w oparciu o opisywane oprogramowanie stworzył automatyczny system odpowiadający za sterowanie elementami inteligentnego domu na podstawie zdarzeń i pomiarów otrzymywanych w czasie rzeczywistym.

GŁÓWNE ZAŁOŻENIA SYSTEMU

Głównym celem implementacji opisywanego rozwiązania było stworzenie prostego narzędzia, które na podstawie zachodzących zdarzeń jest w stanie sterować dedykowanymi urządzeniami jak: oświetlenie, bramy, rolety itp. Całość powinna umożliwić łatwą konfigurację przepływu danych najlepiej w sposób graficzny lub też w innej formie, czytelnej dla użytkownika, a także pozwalać modyfikować logikę, a co za tym idzie zachowanie systemu, bez konieczności pisania kodu oraz kompilacji rozwiązania. Niniejsze założenia zdefiniowano następująco:

- zmiana logiki działania systemu nie wymaga pisania kodu ani kompilacji aplikacji,
- możliwe jest graficzne projektowanie i definiowanie przepływu sygnałów i zdarzeń,
- konfiguracja przepływu danych (ang. workflow) oparta jest na łatwo interpretowanym pliku tekstowym w formacie xml wykorzystującym zdefiniowane „bloki" logiczne,
- każdy z „bloków" jest niezależnym wątkiem uruchamianym na podstawie stanu danych wejściowych,
- poszczególne bloki są połączone, a przepływ danych pomiędzy nimi zależy jedynie od stanu bloku poprzedzającego oraz parametrów wejściowych,
- każdy z bloków logicznych jest reprezentowany przez klasę napisaną w języku java, a przejrzysta struktura klas oraz zdefiniowane interfejsy pozwalają tworzyć własne „bloki" logiczne oraz rozbudowywać istniejące rozwiązania.

SZCZEGÓŁY IMPLEMENTACYJNE

Aby spełnić wyżej wymienione założenia, zaproponowano rozwiązanie, które opiera się na abstrakcyjnej klasie bazowej reprezentującej podstawowy „blok" logiczny.

Listing 1. Klasa bazowa Task

```
public abstract class Task extends Thread {
   public void addConnection(int outputId, Connection connection) {
   public void after()
   public void before()
   public Set<Integer> getAllConnectedOutputs()
   public List<Connection> getAllConnectionsForOutput(int outputId)
   public Object getParameter(int parameterId)
   public Object getValue(int inputNumber) {
   public boolean isRunning()
   public void populateOutputs()
   public void run()
   public void setParameter(int parameterId, Object value)
   public void setRunning(boolean isRunning)
   public void setTaskId(int taskId)
   public void setValue(int inputNumber, Object value)
   private void waitForData()
   public void deleteConnections()
   public String describeClass()
   public String describeFields()
   public abstract boolean isReady();
   public abstract void execute();
}
```

Źródło: opracowanie własne.

Zdefiniowane metody odpowiadają zarówno za wstępną konfigurację „bloku" jak również za przesył danych pomiędzy „blokami". Po stronie użytkownika tworzącego własne rozwiązanie pozostaje zaimplementowanie dwóch abstrakcyjnych metod *isReady()* oraz e*xecute()*.

Metoda *isReady()* wywoływana jest w momencie ustalenia nowej wartości zmiennej wejściowej oraz definiuje warunki, które muszą zostać

spełnione aby „blok" rozpoczął swoje działanie, czyli wywołał metodę *execute()*. Natomiast metoda *execute()*, odpowiada za logikę wykonywanego bloku. To w niej użytkownik definiuje zadania, które mają zostać wykonane przed propagacją sygnału wyjściowego do kolejnego bloku.

Poniżej zaprezentowano implementację przykładowego bloku logiczny reprezentujący operację koniunkcji (operacja AND):

Listing 2 Implementacja bloku logicznego AND

```
@TaskPackage(name = "Logic")
public class And extends Task{
        @Input(id = 1)
        private Boolean input1 = null;
        @Input(id = 2)
        private Boolean input2 = null;
        @Output(id = 1)
        private Boolean output;
        public And(int taskId) {
                super(taskId);
        }
        @Override
        public void execute() {
                output = input1 & input2;
                input 1 = null;
                input2 = null;
        }
        @Override
        public boolean isReady() {
                if ((input1 != null) && (input2 != null)) {
                        return true;
                }
                return false;
        }
}
```

Źródło: opracowanie własne.

Sygnały wejściowe, wyjściowe oraz parametry ustawiane w ramach konfiguracji systemu wykorzystują mechanizm adnotacji. Mechanizm ten pozwala przekazywać dodatkowe metadane, które mogą być wykorzystane na etapie kompilacji, jak również w czasie działania programu. W czasie działania aplikacji skorzystano z mechanizmu refleksji aby odczytać dane klasy. Wymieniony mechanizm odpowiada za znajdowanie pól oznaczających wejścia jak i wyjścia oraz pola definiujące parametry systemu. Dzięki takiemu rozwiązaniu użytkownik w sposób swobodny może definiować zarówno wejścia jak i wyjścia dla każdego z bloku logicznego oraz definiować parametry, które są niezbędne do poprawnego wykonania określonego działania. Całość rozwiązania dopełnia przejrzysta konfiguracja w postaci pliku XML.

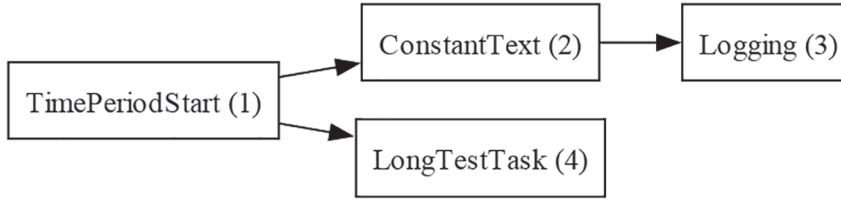
LOGIKA STEROWANA ZDARZENIAMI ORAZ WIELOWĄTKOWOŚĆ

Jak zostało wspomniane wcześniej, jedną z idei przyświecających podczas przygotowania opisywanego rozwiązania, było stworzenie frameworka sterowanego zdarzeniami [9]. Zaletą takiego podejścia jest całkowita asynchroniczność wykonywanych, zdefiniowanych zadań. Każde z nich uruchamiane jest niezależnie, na podstawie stanu wewnętrznego obiektu i danych wejściowych. Pozwala to budować zaawansowane rozwiązania, które łatwo można skalować w zależności od potrzeb. Dodawanie kolejnych elementów nie wpływa, o ile sygnały nie są w jednym torze danych, na wykonywanie innych wcześniej zdefiniowanych zadań. W tym celu zdecydowano, że implementacja każdego z elementów systemu, „bloku" będzie opierała się o wątek w języku java, (ang. *Thread)* [10][11]. Dlatego też podstawowa klasa abstrakcyjna *Task* rozszerza klasę *Tread*. W momencie odczytu pliku xml definiującego logikę systemu, następuje utworzenie i uruchomienie poszczególnych bloków, każdy w osobnym wątku. Następnie po sprawdzeniu stanu wejść oraz warunków uruchomienia wątku, każdy z bloków przechodzi w stan uśpienia. Obudzenie wątku następuje w momencie ustawienia wartości na wejściu wątku, co skutkuje wywołaniem metody *isReady()*, i po wykonaniu zadania, następuje przejście do stanu uśpienia. Podejście takie gwarantuje, że wątki, które są rzadko uruchamiane nie zajmują czasu procesora.

PRZYKŁAD ZASTOSOWANIA OPISANEGO ROZWIĄZANIA

Poniżej zaprezentowano graficzną reprezentację przykładowej aplikacji zbudowanej w oparciu o zdefiniowany, zaprezentowany poniżej schemat logiczny. Aplikacja, co cztery sekundy, w tym celu wykorzystano blok *TimePeriodStart*, loguje – blok *Logging*, określony tekst – zdefiniowany w bloku *ConstantText* na standardowe wyjście (w tym przypadku będzie to ekran komputera) oraz uruchamia blok *LongTestTask*, które symuluje długo wykonywane zadanie (około 7 sek).

Rysunek 2. Schemat blokowy zaimplementowanej przykładowej logiki



Źródło: opracowanie własne.

oraz jej konfiguracja w pliku *xml*:

Listing 3. Przykładowa konfiguracja XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Configuration>
  <task class="lowcode.tasks.time.TimePeriodStart" id="1" >
    <parameters>
      <parameter id="1" value="4000"/>
    </parameters>
    <outputs>
      <output id="1" task="2" input="1" />
    </outputs>
  </task>
```

```
  <task class="lowcode.tasks.text.ConstantText" id="2">
    <parameters>
      <parameter id="1" value="Hello word example"/>
    </parameters>
    <outputs>
      <output id="1" task="3" input="1" />
      <output id="1" task="4" input="1" />
    </outputs>
  </task><!--text -->
  <task class="lowcode.tasks.outputs.Logging" id="3"> <!-- logging -->
  </task>
  <task class="lowcode.tests.LongTestTask" id="4">
  </task>
</Configuration>
```

Źródło: opracowanie własne.

Kod samego programu, napisanego w języku java ogranicza się jedynie do dwóch głównych kroków:

- w pierwszym następuje wczytanie pliku konfigurującego parametry logowania wymagane przez użytą bibliotekę log4j,
- w drugim kroku, pokazanym poniżej następnie wczytanie pliku „tasks.xml" definiującego przepływ danych (opisanego powyżej) i uruchomienie wszystkich zdefiniowanych zadań.

Listing 4. Konfiguracja odczytywania pliku konfiguracyjnego

```
FileReader fileReader = new FileReader("tasks.xml");
fileReader.crateModel(taskManager);
taskManager.startTasks();
```

Źródło: opracowanie własne.

Opisany fragment kodu jest niezależny od logiki zdefiniowanej przez użytkownika w pliku *task.xml*. Oznacza to, że raz skompilowany kod może

być wykorzystywany w prawie dowolnym rozwiązaniu, a zmiany logiki wymagają jedynie zmian w pliku *task.xml*.

Całość aplikacji napisanej w języku java wygląda następująco.

Listing 5. Kod uruchamiający aplikację

```
public class Main {
  public static void main(String[] args) {
    /* Wczytanie parametrów konfiguracyjnych logowania */
      LoggerContext context = (LoggerContext) LogManager.getContext(false);
    File file = new File("log4j2.xml");
    context.setConfigLocation(file.toURI());

    /* Wczytanie zadań i ich uruchomienie */
    TaskManager taskManager = new TaskManager();
    FileReader fileReader = new FileReader("tasks.xml");
    fileReader.crateModel(taskManager);
    taskManager.startTasks();
  }
}
```

Źródło: opracowanie własne.

Po uruchomieniu aplikacji, na ekranie powinniśmy zobaczyć tekst *„Hello word example"*, który wyświetlany jest co cztery sekundy. Poniżej pokazano fragment wyświetlany na ekranie komputera po uruchomieniu aplikacji.

Listing 6. Wiadomości wyświetlane na ekranie komputera

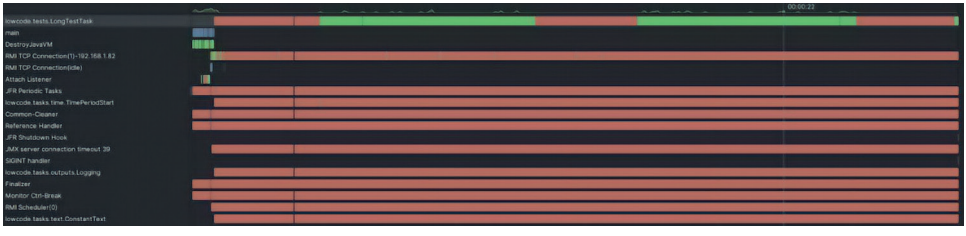
```
09:49:58.097 [main] INFO  lowcode.utils.TaskManager - The start signal has been sent to all tasks
09:49:58.098 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Has been started
09:49:58.098 [lowcode.tests.LongTestTask] INFO  lowcode.tests.LongTestTask - Has been started
09:49:58.098 [lowcode.tasks.text.ConstantText] INFO  lowcode.tasks.text.ConstantText - Has been started
09:49:58.099 [lowcode.tasks.time.TimePeriodStart] INFO  lowcode.tasks.time.TimePeriodStart - Has been
started
```

```
09:50:02.104 [lowcode.tests.LongTestTask] INFO  lowcode.tests.LongTestTask - Started
09:50:02.104 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Hello word example
09:50:06.106 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Hello word example
09:50:10.106 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Hello word example
09:50:10.250 [lowcode.tests.LongTestTask] INFO  lowcode.tests.LongTestTask - Stopped: 8145.0 ms
09:50:14.108 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Hello word example
09:50:14.108 [lowcode.tests.LongTestTask] INFO  lowcode.tests.LongTestTask - Started
09:50:18.108 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Hello word example
09:50:22.109 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Hello word example
09:50:22.405 [lowcode.tests.LongTestTask] INFO  lowcode.tests.LongTestTask - Stopped: 8297.0 ms
09:50:26.109 [lowcode.tasks.outputs.Logging] INFO  lowcode.tasks.outputs.Logging - Hello word example
09:50:26.109 [lowcode.tests.LongTestTask] INFO  lowcode.tests.LongTestTask – Started
```

Źródło: opracowanie własne.

Dla uruchomionej aplikacji wykonano zrzut ekranu, pokazujący poszczególne wątki oraz stan każdego z nich na przestrzeni około 10 sekund. Wyraźnie widać, że większość wątków znajduje się w stanie uśpienia (kolor czerwony), co dotyczy wątków odpowiadających za uruchomienie co 4 sekundy całego procesu *(TimePeriodStart)*, przekazanie tekstu (ConstantText) do bloku logującego *(Logging)* oraz zalogowanie komunikatu na ekranie. Na tym tle wyraźnie jednak zaznacza się blok, kolorem zielonym, wykonujący prace wymagające czasu *(LongTestTask)*. Blok ten po wykonaniu zadania przechodzi w stan uśpienia, co zobrazowane jest kolorem czerwonym, znajdującym się pomiędzy kolorem zielonym.

Rysunek 3. Wątki oraz ich wywołanie w czasie

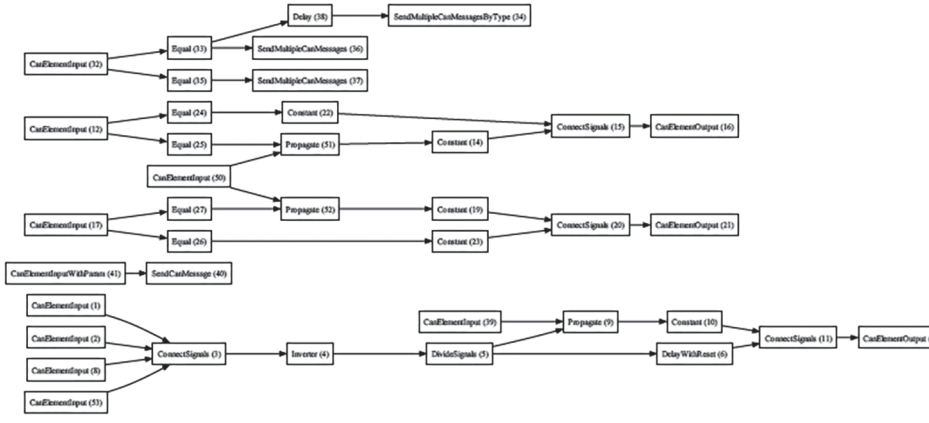


Źródło: opracowanie własne.

Całość zachowuje się więc zgodnie z założeniami. Pojedyncze bloki działają asynchronicznie, a ich konfiguracja odbywa się za pomocą prostego pliku w formacie XML.

Omawiane rozwiązanie zostało z powodzeniem użyte przez Autora – w domu – jako system sterujący inteligentnym budynkiem, odpowiedzialnym za sterowanie oświetleniem garażu. Całość składa się z 45 bloków odpowiedzialnych między innymi za pomiar wartości oświetlenia, informację o otwarciu drzwi oraz bram garażowych, a także odczyt informacji z czujników ruchu. Graficzną reprezentację przepływu danych pomiędzy blokami zaprezentowano poniżej.

Rysunek 4. Graficzna reprezentacja przykładowej logiki sterującej inteligentnym domem



Źródło: opracowanie własne.

POTENCJALNE PROBLEMY

Ze względu na specyfikę działania proponowanego rozwiązania oraz pełną asynchroniczność pomiędzy poszczególnymi zadaniami występują duże problemy z testowaniem całości zdefiniowanej logiki. Poszczególne „bloki" systemu są testowane za pomocą testów jednostkowych, jednak, kompleksowe testy wymagają zagwarantowania przepływu sygnału pomiędzy poszczególnymi blokami. Ze względu na to, że każdy z wątków

wybudzany jest w momencie zmiany stanu jednego z wejść, a następnie sprawdzane są warunki uruchomienia zadania, testowanie całej struktury wymagałoby znajomości stanów każdego z wejść w przestrzeni czasu, bądź też odpowiednie wymuszenie takiego stanu na każdym z bloków. Zatem symulowanie poszczególnych sygnałów lub wykorzystanie bloków o znanych zadanych parametrach (np. generujących określony sygnał w czasie niezależnie od stanu wejść) pozwala z pewnym prawdopodobieństwem ocenić jakość stosowanego rozwiązania.

ZAKOŃCZENIE

Zaprezentowany framework, pomimo zastosowanych standardowych rozwiązań języka java i dzięki prostocie konfiguracji, która umożliwia, w sposób czytelny dla użytkownika definiowanie przepływu danych oraz zachowania systemu, pozwala w sposób prawie nieograniczony na rozbudowę istniejących bloków logicznych lub tworzenie własnych. Podejście takie pozwala zastosować omawiane rozwiązanie w wielu dziedzinach, takich jak przepływ danych i dokumentów, automatyzacja procesów, bądź też sterowanie urządzeniami w szczególności IoT (ang. Internet of Things). Całość może być uruchamiana na zwykłych komputerach klasy PC, bądź też na coraz częściej spotykanych, komputerach jednopłytkowych typu Raspberry-Pi.

BIBLIOGRAFIA

1. https://pl.wikipedia.org/wiki/Platforma_niskokodowa.
2. https://ccit.pl/event-driven-architecture-2/.
3. https://pl.wikipedia.org/wiki/Wielow%C4%85tkowo%C5%9B%C4%87.
4. https://pl.wikipedia.org/wiki/Framework.
5. https://www.parkiet.com/parkiet-plus/art37031971-zapotrzebowanie-na-programistow-juz-siega-zenitu-szczytu-popytu-nie-widac.
6. https://www.precedenceresearch.com/software-market.
7. https://www.globenewswire.com/en/news-release/.
8. https://itwiz.pl/10-trendow-dotyczacych-platform-low-code-na-rok-2023/.
9. Bellemare A. *Mikrousługi oparte na zdarzeniach. Wykorzystanie danych w organizacji na dużą skalę.* Helion S.A. 2021, s. 28.

10. Niemeyer P., Knudsen J. *Java. Wprowadzenie*, Wydawnictwo Helion 2023, s. 199.
11. Eckel B. *Thinking In Java*, Wydawnictwo Helion 2006 s. 907.
12. Gamma E., Helm E., Johnson R., Vlissides J. *Wzorce projektowe*, Wydawnictwo Naukowo-Techniczne, 2008.
13. Hunt A., Thomas D. *Junit Pragmatyczne testy jednostkowe w Javie*, Wydawnictwo Helion 2006.