



POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Programowania

Inżynieria Oprogramowania i Baz Danych

Mateusz Podstawka

32920

Efektywne zarządzanie bibliotekami zewnętrznymi w środowisku IntelliJ IDEA

Praca magisterska
pod kierunkiem naukowym
dr inż. Mariusza Trzaski

Warszawa, Czerwiec, 2026

Streszczenie

Efektywne zarządzanie bibliotekami zewnętrznymi w środowisku IntelliJ IDEA

Niniejsza praca magisterska dotyczy analizy cyklu życia systemów informatycznych pod kątem długu technologicznego wywołanego starzeniem się zewnętrznych bibliotek w ekosystemie języka Java. W ramach badań zweryfikowano współczesne techniki redukcji długu technologicznego oraz opracowano nową koncepcję zarządzania kodem, która łączy analizę struktur składniowych z wnioskowaniem modeli sztucznej inteligencji w celu zmniejszenia nakładu pracy programisty. Zaproponowane podejście automatycznie tworzy rekomendacje naprawcze na podstawie komentarzy Javadoc pochodzących z repozytoriów Maven oraz wprowadza bezpieczne zmiany w kodzie źródłowym po akceptacji użytkownika. Potwierdzeniem wykonalności założeń teoretycznych jest dwumodułowy prototyp oferujący wtyczkę do środowiska IntelliJ IDEA opartą o modele PSI oraz konsolowe narzędzie do masowego audytu kodu bajtowego przy użyciu biblioteki ASM.

Słowa kluczowe: Refaktoryzacja kodu, statyczna analiza kodu, IntelliJ Platform SDK, LLM, zarządzanie zależnościami

Spis treści

1. Wstęp	5
1.1. Kontekst problemu	5
1.2. Cel pracy	5
1.3. Rozwiązanie przyjęte w pracy	6
1.4. Rezultaty pracy	6
1.5. Organizacja pracy	6
2. Podstawy teoretyczne zarządzania cyklem życia oprogramowania	8
2.1. Zjawisko długu technicznego i starzenia się oprogramowania	8
2.2. Ekosystem zależności i piekło zależności	9
2.3. Cykl życia API i polityka deprecjacji	10
2.4. Refaktoryzacja jako element higieny kodu	11
2.5. Analiza stanu sztuki	11
2.5.1. Narzędzia natywne platformy Java (JDK)	11
2.5.2. Systemy statycznej analizy kodu (SAST)	12
2.5.3. Systemy automatycznej refaktoryzacji na przykładzie OpenRewrite	12
2.5.4. Inspekcje w zintegrowanych środowiskach programistycznych (IDE)	14
2.5.5. Wsparcie dużych modeli językowych w procesie migracji	14
2.6. Wnioski z analizy stanu techniki	15
3. Propozycja nowego rozwiązania	16
3.1. Założenia architektoniczne i idea systemu	16
3.2. Specyfikacja wymagań projektowych	18
3.2.1. Komponent środowiska deweloperskiego	18
3.2.2. Niezależny moduł konsolowy	18
3.2.3. Silnik ekstrakcji metadanych i transformacji kodu	18
4. Technologie i narzędzia wykorzystane w rozwiązaniu	19
4.1. Java	19
4.2. Ekosystem IntelliJ Platform SDK	19
4.2.1. Architektura rozszerzeń i cykl życia wtyczki	19
4.2.2. Model Reprezentacji Kodu	20
4.3. Wsparcie dużych modeli językowych	21
4.3.1. Modele chmurowe	21
4.3.2. Lokalne modele generatywne i środowisko uruchomieniowe	22
4.3.3. Inżynieria zapytań	22
4.4. System budowania Gradle	23
5. Implementacja wtyczki do IntelliJ IDEA wspomagającej zarządzanie bibliotekami zewnętrznymi	25
5.1. Architektura systemu i podział na warstwy logiczne	25
5.2. Integracji z ekosystemem Maven	26
5.3. Analiza kodów źródłowych bibliotek zewnętrznych	28
5.4. Skanowania projektu użytkownika przy pomocy modelu PSI	31
5.5. Skanowanie projektu użytkownika przy pomocy biblioteki ASM	33
5.6. Integracja z modelem AI i inżynieria zapytań	38

5.7.	Transformacja kodu i modyfikacja drzewa PSI	41
5.8.	Warstwa prezentacji i interfejsu użytkownika	45
6.	Przykładowe wykorzystanie prototypu	47
6.1.	Wizualizacja głównego interfejsu	47
6.2.	Prezentacja wyników analizy	49
6.2.1.	Weryfikacja wycofanych interfejsów w pakietach zewnętrznych	49
6.2.2.	Identyfikacja przestarzałego kodu w projekcie użytkownika	50
6.3.	Interaktywne mechanizmy naprawcze	51
6.4.	Analiza z wykorzystaniem interfejsu wiersza poleceń	53
6.4.1.	Skanowanie pojedynczej zależności zewnętrznej	53
6.4.2.	Porównywanie wersji zależności	54
6.4.3.	Masowy audyt lokalnego projektu	55
6.4.4.	Wspomaganie użytkownika w trybie interaktywnym	58
7.	Podsumowanie i wnioski końcowe	60
7.1.	Trudności napotkane podczas realizacji projektu	60
7.2.	Mocne i słabe strony rozwiązania	61
7.2.1.	Mocne strony systemu	61
7.2.2.	Słabe strony systemu	61
7.3.	Kierunki dalszego rozwoju projektu	62
7.4.	Zakończenie	63
	Bibliografia	64
	Spis rysunków	66
	Spis listingów	68
	Spis tabel	69

1. Wstęp

Współczesne środowiska programistyczne w dużej mierze opierają się na zewnętrznych bibliotekach i frameworkach. Popularność ekosystemu Maven i rosnąca liczba dostępnych zależności znacząco przyspiesza proces tworzenia oprogramowania, jednakże rodzi również istotne wyzwania związane z zarządzaniem cyklem życia wykorzystywanych komponentów. Jednym z kluczowych problemów jest zarządzanie przestarzałymi metodami (ang. *deprecated methods*). Problem ten nie jest jedynie kwestią techniczną. Wykorzystywanie przestarzałych elementów kodu prowadzi do szeregu konsekwencji, od zwiększenia długu technologicznego, poprzez potencjalne problemy z bezpieczeństwem, aż po trudności w aktualizacji zależności. Badania Sawanta, Robbesa i Bacchelliego [31] wykazały, że większość klientów popularnych bibliotek Java nie reaguje na oznaczenia deprecjacji, co w dłuższej perspektywie może skutkować wzrostem kosztów utrzymania oprogramowania. Z kolei Brito i in. [4] zwracają uwagę, że komunikaty o deprecjacji często nie zawierają jasnych wskazówek dotyczących zastąpienia przestarzałych metod, co dodatkowo utrudnia proces refaktoryzacji. W badaniu Bonordena i Riebischa [3] podkreślono natomiast, że zarządzanie ewolucją API, w tym wykrywanie i obsługa deprecjacji, stanowi jedno z najważniejszych, lecz wciąż niedostatecznie zautomatyzowanych wyzwań w inżynierii oprogramowania.

Niniejsza praca przedstawia analizę cyklu życia współczesnych systemów informatycznych, autorską koncepcję zarządzania długiem technologicznym oraz implementację prototypu zaprojektowanego do automatycznej detekcji i wspomagania refaktoryzacji przestarzałych metod w zależnościach Maven.

1.1. Kontekst problemu

Maven stanowi standard zarządzania zależnościami w projektach Java. Centralne repozytorium Maven obejmuje obecnie ponad 63 miliony artefaktów [34] (stan na początek roku 2026) i obsługuje ponad 1.5 miliarda pobrań rocznie [33]. Każda biblioteka w tym ekosystemie ewoluuje w czasie, wprowadzając nowe funkcjonalności, poprawki błędów oraz oznaczając wybrane elementy API jako przestarzałe. Proces deprecjacji przebiega etapowo, od oznaczenia metod adnotacją `@Deprecated` i odpowiednim znacznikiem JavaDoc, przez dodanie uzasadnienia decyzji oraz wskazanie alternatyw, aż po generowanie ostrzeżeń kompilacji i ostateczne usunięcie metody z API. W praktyce problemem okazuje się brak świadomości deweloperów o tym, że używane przez nich metody zostały oznaczone jako przestarzałe. Zjawisko to jest szczególnie widoczne w przypadku zależności pośrednich, które są wciągane do projektu automatycznie i nie zawsze poddawane ręcznej weryfikacji.

1.2. Cel pracy

Głównym celem niniejszej pracy magisterskiej jest analiza cyklu życia współczesnych systemów informatycznych pod kątem mechanizmów powstawania długu technologicznego. W ramach podjętych badań przewidziano weryfikację współczesnych technik radzenia sobie z tym zjawiskiem, co obejmuje identyfikację mocnych i słabych stron poszczególnych rozwiązań oraz ocenę ich efektywności w praktycznym zastosowaniu. Na podstawie uzyskanych wyników przedstawiona zostanie własna koncepcja

zarządzania tym problemem w codziennej praktyce inżynierskiej, która wprowadza usprawnienia w stosunku do istniejących narzędzi. W celu potwierdzenia wykonalności zaproponowanych założeń teoretycznych opracowany zostanie dwumodułowy prototyp systemu informatycznego.

1.3. Rozwiązanie przyjęte w pracy

Realizację prototypu oparto na zestawie sprawdzonych i powszechnie stosowanych w przemyśle informatycznym technologii, do których należą następujące komponenty:

- język programowania **Java** w wersji 17 zapewniającej długoterminowe wsparcie oraz wysoką stabilność uruchomieniową,
- oficjalny pakiet narzędziowy **IntelliJ Platform SDK** dedykowany do budowania zaawansowanych rozszerzeń oraz integracji z aplikacją macierzystą,
- mechanizmy inspekcji struktur językowych oraz analizy skompilowanych plików klas, reprezentowane przez model **PSI** oraz niskopoziomą bibliotekę **ASM**,
- współczesne chmurowe usługi sztucznej inteligencji wykorzystujące komercyjne modele językowe do generowania rekomendacji refaktoryzacyjnych,
- elastyczne środowisko uruchomieniowe **Ollama** pozwalające na bezpieczne i w pełni autonomiczne wykonywanie lokalnych modeli generatywnych bezpośrednio na maszynie dewelopera bez potrzeby transferu kodu poza sieć wewnętrzną.

1.4. Rezultaty pracy

Rezultatem pracy jest analiza cyklu życia oprogramowania pod kątem powstawania długu technologicznego oraz zaproponowanie nowej koncepcji zarządzania tym zjawiskiem. W ramach pracy zbadano procesy starzenia się kodu źródłowego wynikające ze starzenia się zewnętrznych bibliotek, a także przeanalizowano ograniczenia istniejących narzędzi automatyzacji.

Na podstawie wyciągniętych wniosków opracowano model ograniczania długu technologicznego, który łączy analizę struktur składniowych kodu z wnioskowaniem modeli sztucznej inteligencji. Zaproponowane podejście ma na celu zmniejszenie nakładu pracy programisty podczas aktualizacji systemów.

Praktycznym potwierdzeniem opracowanej koncepcji jest dwumodułowy system informatyczny wspierający utrzymanie spójności architektury oprogramowania poprzez zautomatyzowaną identyfikację oraz eliminację przestarzałych elementów interfejsów programistycznych. Pierwszy z modułów stanowi interaktywna wtyczka do środowiska **IntelliJ IDEA** umożliwiająca analizę kodu za pomocą struktur **PSI** oraz bezpieczną refaktoryzację pod nadzorem programisty. Drugi komponent to niezależne narzędzie wiersza poleceń służące do masowego audytu plików klas przy użyciu niskopoziomowej biblioteki **ASM** oraz generowania raportów w formacie **JSON**. Implementacja tego systemu służy jako dowód, że teoretyczne założenia dotyczące sprawniejszego eliminowania przestarzałych elementów są możliwe do zrealizowania i pozwalają na skrócenie czasu potrzebnego na utrzymanie kodu.

1.5. Organizacja pracy

Treść niniejszej pracy magisterskiej została podzielona na siedem rozdziałów, które odzwierciedlają proces realizacji założeń badawczych oraz prac programistycznych nad systemem. W skład struktury dokumentu wchodzi następujące części:

- **Rozdział pierwszy** – wprowadza w tematykę zarządzania cyklem życia zależności, określa tło problematyki długu technologicznego oraz definiuje główne cele i uzyskane rezultaty inżynierskie,
- **Rozdział drugi** – zawiera teoretyczne podwaliny procesów starzenia się oprogramowania oraz prezentuje krytyczną analizę istniejących systemów statycznej weryfikacji kodu, narzędzi automatycznej refaktoryzacji i modeli językowych,
- **Rozdział trzeci** – koncentruje się na omówieniu fundamentalnych założeń architektonicznych systemu oraz szczegółowej specyfikacji wymagań projektowych dla komponentu graficznego i modułu konsolowego,
- **Rozdział czwarty** – charakteryzuje wybrany stos technologiczny ze szczególnym uwzględnieniem specyfiki środowiska *IntelliJ Platform SDK* oraz koncepcji integracji z modelami generatywnymi sztucznej inteligencji,
- **Rozdział piąty** – opisuje techniczne szczegóły implementacji warstw logicznych aplikacji, przedstawiając algorytmy parsujące *PSI*, metody skanowania kodu bajtowego za pomocą biblioteki *ASM* oraz strukturę modułu transformacji kodu,
- **Rozdział szósty** – ilustruje praktyczne scenariusze wykorzystania gotowego prototypu w codziennej pracy deweloperskiej, prezentując działanie interfejsu graficznego, automatyczne akcje naprawcze oraz możliwości wiersza poleceń,
- **Rozdział siódmy** – podsumowuje zrealizowane zadania badawcze, zawiera analizę napotkanych trudności realizacyjnych, bilans mocnych i słabych stron technologii oraz wytycza kierunki dalszego rozwoju oprogramowania.

2. Podstawy teoretyczne zarządzania cyklem życia oprogramowania

Niniejszy rozdział wprowadza teoretyczny oraz narzędziowy kontekst problemu zarządzania zależnościami w projektach Java. W pierwszej kolejności zdefiniowano pojęcia długu technicznego, cyklu życia API oraz roli refaktoryzacji. Następnie przeprowadzono przegląd aktualnego stanu techniki, obejmujący narzędzia natywne, systemy statycznej analizy, systemy automatycznej analizy, inspekcje w IDE oraz modele językowe. Rozdział zamyka zestawienie ograniczeń omawianego oprogramowania, co stanowi bezpośrednie uzasadnienie dla koncepcji autorskiego systemu zaproponowanego w niniejszej pracy.

2.1. Zjawisko długu technicznego i starzenia się oprogramowania

Koncepcja długu technicznego (ang. *technical debt*) została po raz pierwszy sformułowana przez Warda Cunninghama w 1992 roku jako metafora finansowa opisująca konsekwencje podejmowania suboptymalnych decyzji projektowych w celu przyspieszenia procesu dostarczania oprogramowania [5]. Zaciąganie długu technicznego objawia się poprzez implementację rozwiązań tymczasowych, brak testów jednostkowych lub świadome użycie przestarzałych technologii.

Podobnie jak w świecie finansów, dług techniczny generuje *odsetki*. W inżynierii oprogramowania przyjmują one postać zwiększonego nakładu pracy wymaganego do wprowadzenia nowych funkcjonalności w przyszłości. Niespłacany dług prowadzi do zjawiska starzenia się oprogramowania (ang. *software aging*) [29]. Kod staje się sztywny, trudny w modyfikacji i niezrozumiały dla nowych członków zespołu, powoli przekształcając się w tzw. system *legacy*. W literaturze przedmiotu kod *legacy* definiuje się często nie przez pryzmat jego wieku, lecz jako kod pozbawiony odpowiedniego pokrycia testami oraz trudny do bezpiecznej refaktoryzacji [6]. Koszty utrzymania takich systemów z czasem zaczynają pochłaniać większość budżetu projektowego, drastycznie ograniczając możliwości innowacyjne przedsiębiorstwa.

Współcześnie szczególną formą długu technicznego, kluczową z perspektywy utrzymania nowoczesnych aplikacji, jest tzw. dług zależności (ang. *dependency debt*). W dobie systemów korzystających z dziesiątek zewnętrznych pakietów, oprogramowanie ulega starzeniu nie tylko w wyniku wewnętrznej degradacji kodu, ale również na skutek stagnacji względem otaczającego go ekosystemu. Nawet aplikacja o poprawnie zaprojektowanej architekturze staje się systemem *legacy*, gdy wykorzystywane przez nią biblioteki tracą wsparcie twórców (ang. *End of Life*) lub gdy odkrywane są w nich luki bezpieczeństwa.

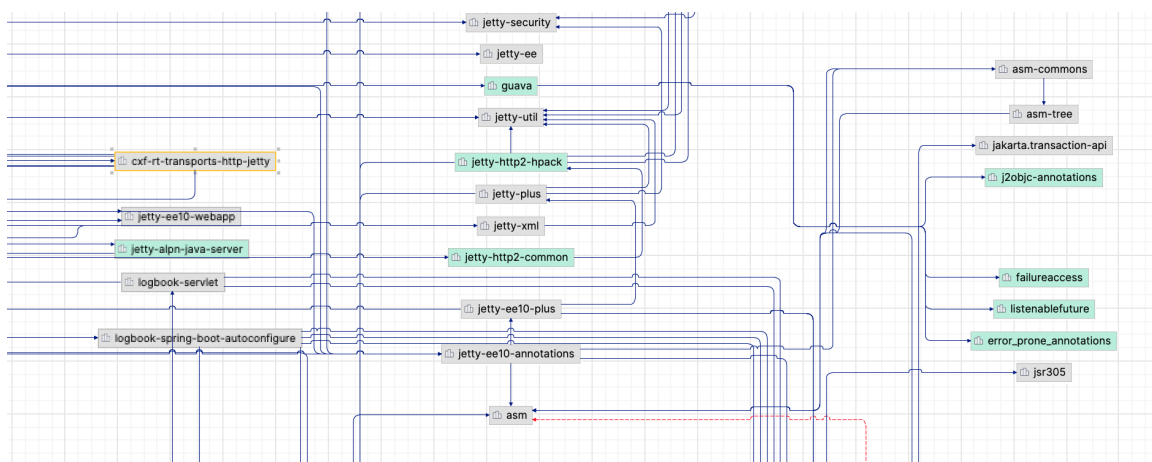
Skalę tego problemu dobrze obrazuje Rysunek 2.1, prezentujący wycinek repozytorium Maven Central dla zależności *spring-core*. Pokazuje on, jak utrzymywanie oprogramowania na starszych, nieaktualnych gałęziach wersji (np. 6.0.x lub 6.1.x) bezpośrednio wystawia system na znane podatności (ang. *vulnerabilities*). Konieczność ich eliminacji wymusza na zespołach programistycznych nieustanne dostosowywanie bazy kodowej do najnowszych wydań API, co nierozdzielnie łączy problematykę higieny kodu z wyzwaniami zarządzania zależnościami.

6.2.x	6.2.18		87	Apr 17, 2026
	6.2.17		161	Mar 13, 2026
	6.2.16		149	Feb 12, 2026
6.1.x	6.1.21	1 vulnerability	78	Jun 12, 2025
	6.1.20	1 vulnerability	67	May 15, 2025
	6.1.19	1 vulnerability	86	Apr 17, 2025
6.0.x	6.0.23	1 vulnerability	83	Aug 14, 2024
	6.0.22	1 vulnerability	75	Jun 13, 2024
	6.0.21	1 vulnerability	46	May 22, 2024

Rysunek 2.1: Zestawienie wersji biblioteki *spring-core* w repozytorium Maven Central, obrazujące problem luk w zabezpieczeniach w przestarzałych gałęziach. Źródło: [23].

2.2. Ekosystem zależności i piekło zależności

Współczesne systemy informatyczne rzadko tworzone są od podstaw. Zgodnie z paradygmatem ponownego wykorzystania kodu (ang. *code reuse*), programiści opierają swoje rozwiązania na gotowych modułach i bibliotekach np. udostępnianych w repozytorium Maven Central. Prowadzi to do powstania złożonego grafu zależności (ang. *dependency graph*), w którym węzłami są pakiety oprogramowania, a krawędziami relacje między nimi. Przykładowy wycinek takiego grafu, obrazujący stopień skomplikowania powiązań w nowoczesnych projektach, przedstawiono na Rysunku 2.2.

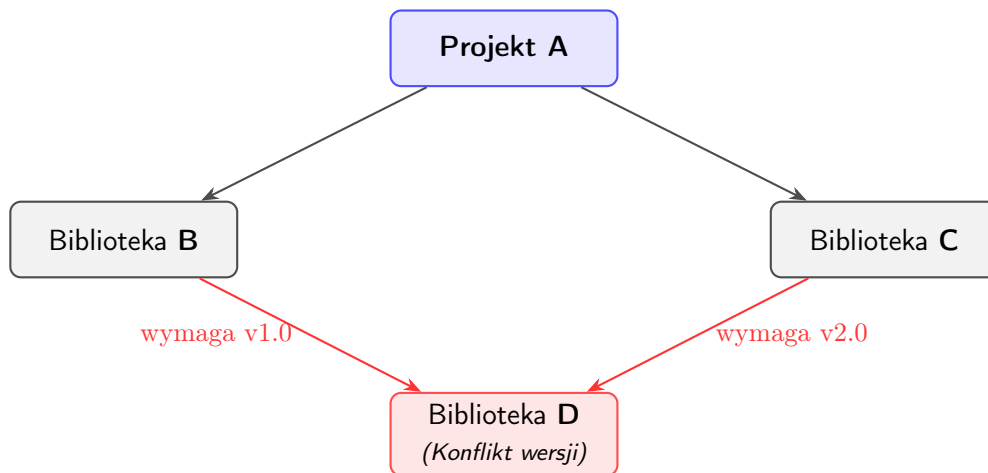


Rysunek 2.2: Wycinek wygenerowanego grafu zależności w środowisku Intelij IDEA. Źródło: opracowanie własne.

Szczególnym wyzwaniem w tym modelu są zależności przechodnie (ang. *transitive dependencies*). Jeśli projekt *A* wymaga biblioteki *B*, a biblioteka *B* wewnętrznie korzysta z biblioteki *C*, to projekt *A* w sposób niejawni staje się zależny również od *C*. Brak ścisłej kontroli nad grafem przechodnim prowadzi do zjawiska znanego w branży jako *piekło zależności* (ang. *Dependency Hell*).

Typowym objawem tego problemu jest konflikt diamentowy (ang. *diamond problem*). Występuje on, gdy dwie bezpośrednie zależności projektu *B* i *C* wymagają dwóch różnych, niekompatybilnych ze sobą wersji tej samej biblioteki bazowej *D*. Rozwiązywanie tego typu konfliktów wymaga precyzyj-

nego zarządzania wersjami oraz cyklicznych aktualizacji całego stosu technologicznego, co w dużych systemach jest procesem wysoce ryzykownym. Graficzną reprezentację tego zjawiska przedstawiono na Rysunku 2.3.



Rysunek 2.3: Schemat powstawania konfliktu diamentowego w grafie zależności. Źródło: opracowanie własne.

2.3. Cykl życia API i polityka deprecjacji

Zewnętrzne biblioteki komunikują się z kodem aplikacji za pomocą interfejsu programistycznego *API*. API stanowi ścisły kontrakt między twórcą biblioteki a jej użytkownikiem. Wraz z ewolucją oprogramowania, twórcy bibliotek zmuszeni są do wprowadzania zmian w tym kontrakcie. Takie zmiany pozwalają na optymalizacji algorytmów, łatania luk bezpieczeństwa lub zmiany sygnatur metod.

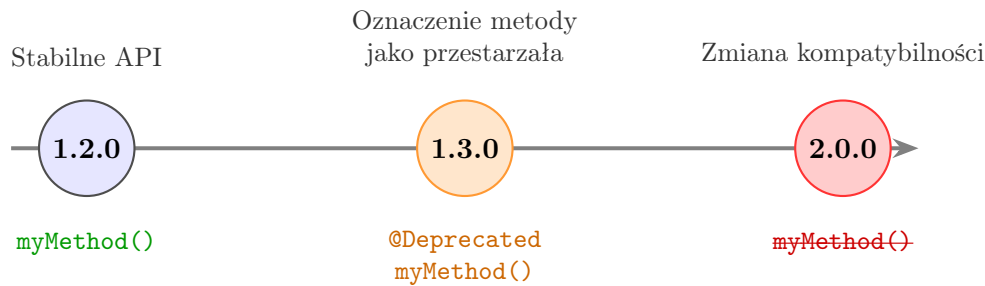
Aby uniknąć nagłego zepsucia kodu (ang. *breaking changes*) u tysięcy użytkowników, stosuje się politykę deprecjacji. W języku Java służy do tego adnotacja `@Deprecated`, która sygnalizuje, że dany element - klasa, metoda, pole - jest przestarzały i nie powinien być używany w nowo pisanym kodzie, ponieważ zostanie usunięty w przyszłych wydaniach. Od wersji Java 9 wprowadzono dodatkowy atrybut `forRemoval=true` [19]. Okres deprecjacji to okno czasowe, w którym programiści mają czas na migrację do rekomendowanego, nowego rozwiązania.

Proces ten jest ściśle powiązany z Semantycznym Wersjonowaniem - **SemVer** [30] (ang. *Semantic Versioning*). Zgodnie ze standardem **SemVer**, numer wersji składa się z trzech segmentów: MAJOR.MINOR.PATCH np. 2.1.4. Zmiany te oznaczają:

- MAJOR - wprowadzenie zmian niekompatybilnych wstecz,
- MINOR - dodanie nowej funkcjonalności,
- PATCH - poprawki błędów, które nie wpływają na działanie API.

Przebieg tego cyklu w praktyce zilustrowano na Rysunku 2.4.

Ścisłe przestrzeganie standardu **SemVer** przez twórców bibliotek stanowi fundament dla wszelkich narzędzi automatyzujących refaktoryzację, ponieważ pozwala w sposób deterministyczny przewidzieć, czy aktualizacja zależności wymusi modyfikację kodu w docelowym projekcie.



Rysunek 2.4: Cykl życia interfejsu API w kontekście Semantycznego Wersjonowania. Źródło: opracowanie własne.

2.4. Refaktoryzacja jako element higieny kodu

Proces spłacania długu technicznego, w tym dostosowywania bazy kodu do nowych wersji API i eliminacji użycia przestarzałych metod, realizowany jest poprzez refaktoryzację. Martin Fowler, pionier w tej dziedzinie, definiuje refaktoryzację jako proces modyfikacji wewnętrznej struktury programu, którego celem jest ułatwienie jego zrozumienia i obniżenie kosztów modyfikacji, bez zmiany jego obserwowalnego zachowania z zewnątrz [7]. Na problem ten wskazuje także Mariusz Trzaska w książce pod tytułem *Modelowanie i implementacja systemów informatycznych 2.0*, w sekcji poświęconej pielęgnacji oraz konserwacji oprogramowania [35].

W kontekście zarządzania zależnościami, ręczna refaktoryzacja tysięcy wywołań przestarzałego API w projektach klasy Enterprise jest procesem żmudnym, kosztownym i podatnym na błędy ludzkie (ang. *error-prone*). Wymaga ona szczegółowej analizy notatek wydawniczych (ang. *release notes*) dla każdej aktualizowanej biblioteki. Z tego względu kluczowym elementem zachowania higieny kodu staje się automatyzacja transformacji kodu, pozwalająca na masowe, deterministyczne lub wspomagane sztuczną inteligencją aplikowanie zmian architektonicznych, co stanowi główny przedmiot badań w kolejnych rozdziałach niniejszej pracy.

2.5. Analiza stanu sztuki

Przeprowadzona analiza literatury oraz dostępnych narzędzi w ekosystemie Java pozwala na wyodrębnienie głównych kategorii rozwiązań, różniących się podejściem do detekcji, poziomem automatyzacji oraz integracją z procesem wytwórczym (ang. *Systems development life cycle*). W niniejszym podrozdziale omówiono reprezentatywne rozwiązania dla każdej z zidentyfikowanych grup, wskazując ich ograniczenia oraz przykłady użycia.

2.5.1. Narzędzia natywne platformy Java (JDK)

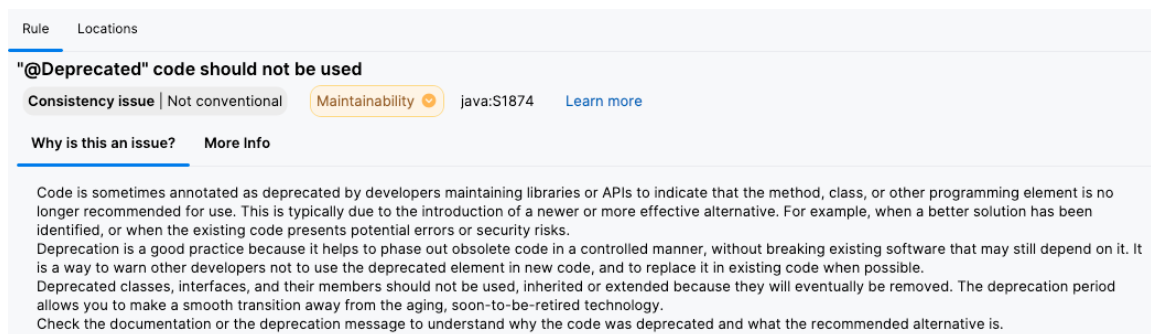
Podstawowym mechanizmem weryfikacji zgodności API jest kompilator `javac`. Przy użyciu flagi `-Xlint:deprecation`, emituje on ostrzeżenia podczas procesu translacji kodu źródłowego. Mechanizm ten ogranicza się do sygnalizacji wystąpienia problemu, nie dostarczając kontekstu architektonicznego ani propozycji migracji [27].

W celu rozszerzenia możliwości diagnostycznych, w wersji Java 9 wprowadzono narzędzie `jdeprscan`, umożliwiające statyczną analizę plików bajtowych - `.class`, `.jar` - bez dostępu do kodu źródłowego. Mimo że rozwiązanie to ułatwia weryfikację gotowych artefaktów, jego zastosowanie w zautomatyzowanych procesach CI/CD napotyka na istotne bariery. Przede wszystkim narzędzie to koncentruje się na samym fakcie użycia przestarzałego elementu, pomijając szerszy kontekst semantyczny, niezbędny do oceny ryzyka refaktoryzacji. Co więcej, generowane raporty mają postać tekstową, co utrudnia ich

maszynowe przetwarzanie i agregację w systemach monitoringu jakości [28].

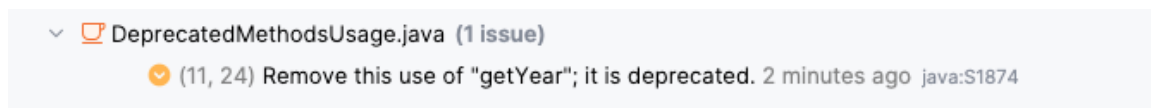
2.5.2. Systemy statycznej analizy kodu (SAST)

Platformy klasy Enterprise, takie jak SonarQube czy wtyczki z rodziny SonarLint [32], podchodzą do problemu deprecjacji w sposób sformalizowany, traktując adnotację `@Deprecated` jako jedną z reguł jakościowych. Naruszenie tej zasady jest kategoryzowane jako dług techniczny lub *Code Smell*, co szczegółowo opisano w dokumentacji i zobrazowano na Rysunku 2.5.



Rysunek 2.5: Widok reguły dotyczącej użycia przestarzałego kodu w interfejsie systemu SonarQube. Źródło: opracowanie własne.

Systemy te działają zazwyczaj w trybie tylko do odczytu - identyfikują problem i prezentują go na dedykowanym dashboardzie, lecz nie integrują się bezpośrednio ze środowiskiem pracy programisty w sposób umożliwiający natychmiastową korektę. Przykładowy raport z takiej detekcji przedstawiono na Rysunku 2.6.



Rysunek 2.6: Przykład wykrycia metody przestarzałej przez system SonarQube. Źródło: opracowanie własne.

Dodatkowo, w kontekście rozległych systemów typu *legacy*, narzędzia te mają tendencję do generowania dużej liczby fałszywych pozytywnych (ang. *false positives*), co w dłuższej perspektywie prowadzi do zjawiska zmęczenia alarmami (ang. *alert fatigue*) i ignorowania ostrzeżeń przez zespoły deweloperskie.

2.5.3. Systemy automatycznej refaktoryzacji na przykładzie OpenRewrite

Istotnym elementem w dziedzinie modernizacji systemów Java jest projekt OpenRewrite. Jest to narzędzie klasy *Automated Refactoring*, które umożliwia masowe przekształcanie kodu źródłowego przy użyciu bezstratnych drzew semantycznych (ang. *Lossless Semantic Trees* – LST). W przeciwieństwie do klasycznych analizatorów opartych na abstrakcyjnych drzewach składniowych (ang. *Abstract Syntax Tree* – AST), modele LST zachowują oryginalne formatowanie kodu np. spacje, wcięcia, co pozwala na automatyczne aplikowanie zmian bezpośrednio do repozytorium [22].

Mechanizm działania OpenRewrite opiera się na podejściu deklaratywnym. Reguły transformacji, nazywane potocznie przepisami (ang. *recipes*), definiują sekwencje operacji modyfikujących składnię i semantykę kodu. Przykłady wbudowanych reguł transformacji przedstawiono na Rysunku 2.7.

Integracja narzędzia z procesem wytwórczym odbywa się zazwyczaj poprzez systemy budowania. Listing 2.1 prezentuje przykładową konfigurację pluginu dla narzędzia Maven, niezbędną do uruchomienia procesu analizy.

Spring Boot 3.x

Composite Recipes

Recipes that include further recipes, often including the individual recipes below.

- [Change `com.datastax.oss` to `org.apache.cassandra`](#)
- [Migrate Enabled to Access Spring Boot Properties](#)
- [Migrate Spring Boot properties to 3.0](#)
- [Migrate Spring Boot properties to 3.1](#)
- [Migrate Spring Boot properties to 3.2](#)
- [Migrate Spring Boot properties to 3.3](#)

Rysunek 2.7: Zestawienie przepisów dla framework'a Spring Boot dostępnych w OpenRewrite. Źródło: [26].

Listing 2.1: Konfiguracja narzędzia OpenRewrite w pliku `pom.xml`. Źródło: opracowanie własne.

```
1 <plugin>
2   <groupId>org.openrewrite.maven</groupId>
3   <artifactId>rewrite-maven-plugin</artifactId>
4   <version>5.30.0</version>
5 </plugin>
```

Po zdefiniowaniu zależności środowiskowej, konieczne jest wskazanie konkretnego przepisu migracyjnego, który zostanie zaaplikowany w projekcie poprzez dodanie go do projektu w postaci pliku `.yml` (zob. Rysunek 2.8).



Rysunek 2.8: Aktywacja narzędzia OpenRewrite z poziomu IDE. Źródło: opracowanie własne.

Ekosystem OpenRewrite oferuje wiele wektorów integracji. Zmiany mogą być aplikowane za pomocą wtyczki do środowiska IntelliJ IDEA lub poprzez wykorzystanie niezależnego interfejsu wiersza poleceń, co prezentuje Listing 2.2.

Listing 2.2: Instalacja i wywołanie narzędzia OpenRewrite z poziomu interfejsu wiersza poleceń. Źródło: opracowanie własne

```
1 brew install openrewrite/rewrite/rewrite
2 rewrite run --recipe org.openrewrite.java.migrate.UpgradeToJava17
```

Mimo zaawansowania technologicznego, rozwiązanie to posiada istotne ograniczenia z punktu widzenia elastycznego zarządzania długiem technicznym. Obecna architektura OpenRewrite opiera się wyłącznie na predefiniowanych, deterministycznych schematach. Narzędzie nie posiada zdolności adaptacyjnych pozwalających na autonomiczną refaktoryzację bibliotek i API, dla których społeczność lub twórcy nie przygotowali wcześniej gotowego przepisu.

Dodatkowym wyzwaniem jest ergonomia pracy. Integracja ze środowiskiem IDE jest często nieintuicyjna i wymaga tworzenia zewnętrznych plików konfiguracyjnych, co w przypadku mniejszych migracji prowadzi do zanieczyszczania struktury projektu zbędnymi artefaktami. Cechy te stanowią barierę w zwinnym i lokalnym testowaniu migracji przez programistów w ich codziennym cyklu pracy.

2.5.4. Inspekcje w zintegrowanych środowiskach programistycznych (IDE)

Zupełnie inne podejście reprezentują środowiska zintegrowane, takie jak IntelliJ IDEA. Wykorzystując zaawansowany model PSI (*Program Structure Interface*), IDE buduje w pamięci wirtualną reprezentację składni i semantyki kodu. Pozwala to na precyzyjne wykrywanie wywołań przestarzałych metod w czasie rzeczywistym oraz oferowanie kontekstowych sugestii naprawy (zob. Rysunek 2.9).



Rysunek 2.9: Wizualizacja inspekcji statycznej w środowisku IntelliJ IDEA. Źródło: opracowanie własne.

Uruchomienie inspekcji wymaga załadowania pełnego modelu projektu oraz zbudowania indeksów, co w trybie headless - bez interfejsu graficznego - jest procesem wysoce zasobożernym, obciążającym procesor i pamięć RAM. Ponadto logika inspekcji jest nierozdzielnie związana z API samej platformy IntelliJ, co uniemożliwia jej proste wyeksportowanie do postaci lekkiego narzędzia wiersza poleceń, mogącego działać niezależnie od instalacji IDE [15].

2.5.5. Wsparcie dużych modeli językowych w procesie migracji

Alternatywą stają się asystenci oparci na dużych modelach językowych, tacy jak GitHub Copilot. Choć oferują oni dużą elastyczność, ich kluczową wadą w kontekście migracji systemów krytycznych jest niedeterminizm. Modele generatywne mogą proponować rozwiązania syntaktycznie poprawne, lecz błędne semantycznie, tzw. halucynacje. Dodatkowo, brak gwarancji spójności zmian w skali wielu modułów jednocześnie utrudnia stosowanie sztucznej inteligencji do masowych i bezpiecznych aktualizacji API w dużych repozytoriach [10].

Sytuację tę potwierdza opublikowany w maju 2026 roku przegląd literatury naukowej, który objął analizę 50 wyselekcjonowanych badań nad automatyzacją refaktoryzacji [20]. Z wyciągniętych przez badaczy wniosków wynika, że najpopularniejszym obiektem analiz w światowych projektach był właśnie język Java, a bazy danych do testów pobierano z platform takich jak GitHub oraz Apache. Z badań wynika, że Odpowiednie konstruowanie zapytań, na przykład metodami One-Shot lub Chain-of-Thought, pozwalało w niektórych przypadkach zmniejszyć zagęszczenie złych praktyk w kodzie nawet o 89%, głównie przy upraszczaniu zbyt długich metod i klas. Pomimo tak obiecujących wyników autorzy przeglądu jednoznacznie wskazują na poważne ograniczenia techniczne sztucznej inteligencji. Modele językowe regularnie generują błędny kod, zupełnie nie radzą sobie ze złożonymi transformacjami, a także często pomijają kluczowe instrukcje refaktoryzacji lub opacznie interpretują intencje programisty. To dowodzi, że samo użycie sztucznej inteligencji bez zewnętrznej kontroli statycznej i sztywnych filtrów weryfikacyjnych jest niewystarczające do bezpiecznej pracy z kodem.

2.6. Wnioski z analizy stanu techniki

Przeprowadzona w niniejszym rozdziale analiza stanu techniki wskazuje na wyraźną lukę narzędziową w procesie wykrywania oraz aktualizacji zewnętrznych zależności. Obecnie dostępne rozwiązania wymuszają na zespołach programistycznych kompromis polegający na wyborze między bezpieczeństwem opartym na sztywnych i trudnych w utrzymaniu regułach a elastycznością modeli generatywnych, która bez odpowiedniego nadzoru deweloperskiego jest obciążona ryzykiem niedeterminizmu oraz błędów składniowych.

W odpowiedzi na te wyzwania w kolejnym rozdziale niniejszej pracy zostanie szczegółowo zaprezentowana autorska koncepcja nowego narzędzia informatycznego adresującego ten problem.

3. Propozycja nowego rozwiązania

W treści tego rozdziału przedstawiono koncepcję autorskiego rozwiązania ukierunkowanego na eliminację trudności wykazanych w toku wcześniejszych analiz oraz sformułowano szczegółowe kryteria, jakim musi sprostać budowane narzędzie informatyczne.

3.1. Założenia architektoniczne i idea systemu

Mając na uwadze ograniczenia analizowane w poprzednich rozdziałach, w ramach niniejszej pracy proponuje się kompleksowe środowisko narzędziowe, którego celem jest uproszczenie lokalizacji przestarałych komponentów i zautomatyzowanie generowania poprawek deweloperskich. Kluczową częścią tej koncepcji jest pozostawienie inżynierowi oprogramowania ostatecznej, interaktywnej kontroli nad sugerowanymi zmianami oraz transakcyjne zabezpieczenie struktury modyfikowanych plików. W ramach projektowanego rozwiązania zakłada się, że system nie będzie opierał się wyłącznie na wiedzy generatywnej sztucznej inteligencji, która w przypadku złożonych transformacji refaktoryzacyjnych bywa niekompletna lub błędna. Zamiast tego proces transformacji ma być zasilany bezpiecznym kontekstem w postaci informacji zawartych w adnotacjach oraz dokumentacji Javadoc pobieranej z zewnętrznych pakietów.

Aby zrealizować założony cel, projektuje się prototypowy system wspomagający zarządzanie bibliotekami zewnętrznymi, w skład którego wchodzi trzy powiązane elementy:

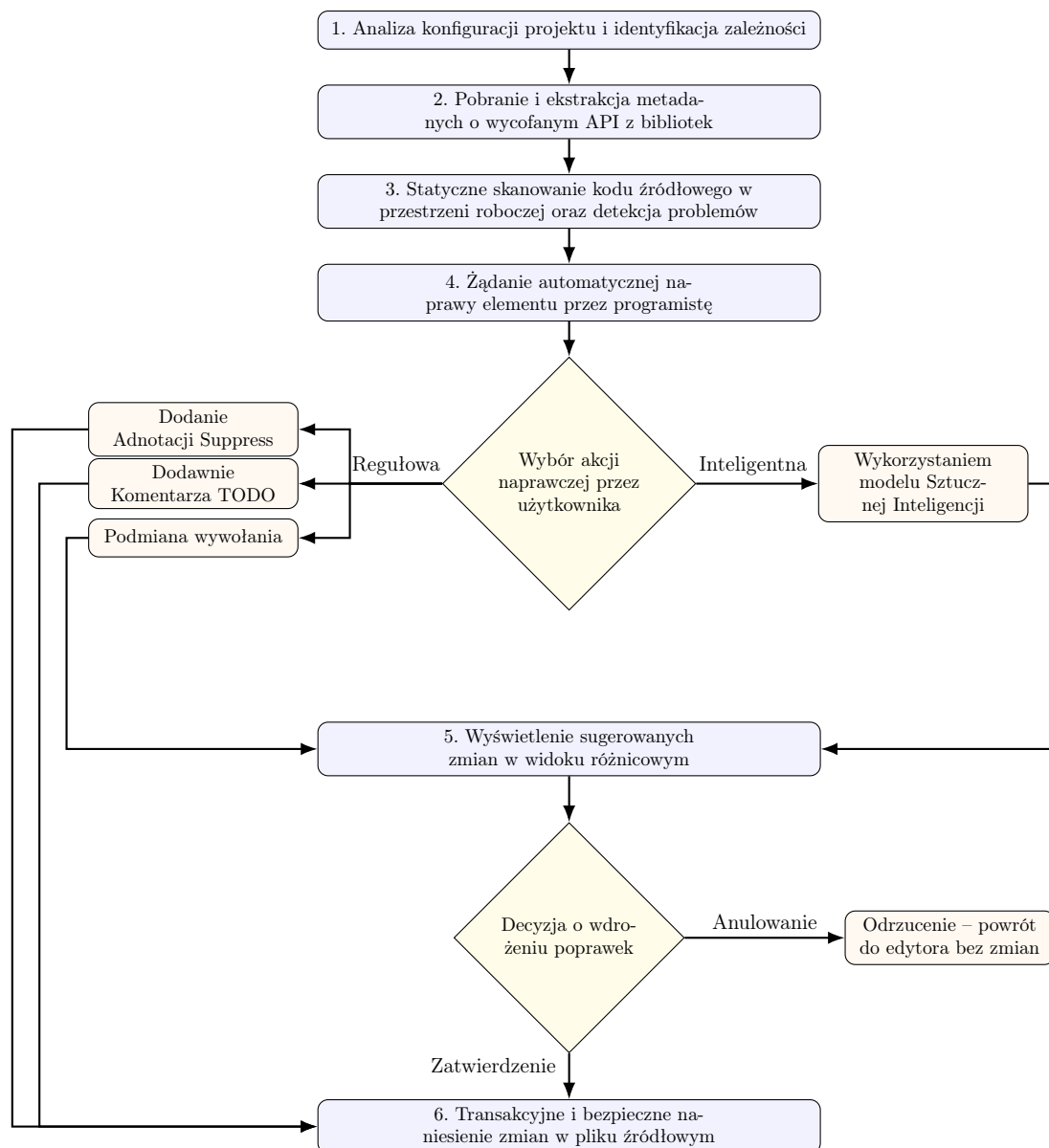
- wtyczka dedykowana dla głównego programu programistycznego dewelopera, która funkcjonuje bezpośrednio w przestrzeni roboczej i udostępnia graficzny interfejs użytkownika, zapewniając bieżące skanowanie otwartych projektów pod kątem niezalecanych zależności oraz interaktywną prezentację poprawek w widoku różnicowym,
- niezależny moduł konsolowy uruchamiany w trybie tekstowym, zaprojektowany z myślą o integracji z platformami automatycznego budowania projektów i ciągłego wdrażania, co pozwala na masową weryfikację kodu bezpośrednio na zewnętrznych serwerach kompilacyjnych,
- silnik analizy zależności oraz transformacji kodu, który odpowiada za automatyczne skanowanie zewnętrznych bibliotek i pobieranie metadanych z dokumentacji Javadoc, a następnie realizuje naprawy deterministyczne na bazie odnalezionych reguł lub generatywne przy użyciu sztucznej inteligencji.

Główny komponent przeznaczony do bezpośredniej interakcji z deweloperem został zaprojektowany jako wtyczka do środowiska IntelliJ IDEA. O wyborze tej konkretnej platformy jako bazy dla interfejsu graficznego zadecydowały następujące czynniki:

- **Bliskość środowiska pracy programisty** – wtyczka integruje się bezpośrednio z głównym programem deweloperskim, eliminując konieczność przełączania na zewnętrzne narzędzia. System jest dostępny dokładnie w momencie i w miejscu, w którym programista pisze kod, co znacząco obniża barierę jego użycia.

- **Wykorzystanie gotowego ekosystemu** – platforma IntelliJ udostępnia rozbudowane pakiety deweloperskie określane jako Software Development Kit, w tym interfejs Program Structure Interface oraz mechanizmy bezpiecznych transakcji na kodzie znane jako Write Action. Pozwala to uniknąć konieczności samodzielnego implementowania złożonych parserów języka Java.
- **Wysoka adopcja rynkowa** – środowisko IntelliJ IDEA jest obecnie najpopularniejszym wyborem wśród programistów pracujących w ekosystemie języka Java [16], co gwarantuje szeroki potencjalny zasięg opracowanego narzędzia.

Koncepcja działania ścieżki z wykorzystaniem interfejsu graficznego systemu została zaprezentowana na Rysunku 3.1. Na zamieszczonym schemacie można zaobserwować, że architekturę całego systemu projektuje się w sposób ściśle interaktywny, co ma gwarantować pełną kontrolę programisty nad modyfikowanym kodem źródłowym.



Rysunek 3.1: Koncepcyjny schemat blokowy przepływu danych i logiki decyzji we wtyczce. Źródło: opracowanie własne.

3.2. Specyfikacja wymagań projektowych

Niniejszy podrozdział zawiera szczegółowe zestawienie kryteriów stawianych przed budowanym systemem informatycznym.

3.2.1. Komponent środowiska deweloperskiego

Wtyczka do głównego programu programistycznego ma za zadanie ułatwić inżynierom oprogramowania bieżące monitorowanie długu technologicznego w trybie interfejsu graficznego. Element ten musi spełniać następujące kryteria:

- bezproblemowe działanie wewnątrz obszaru roboczego programisty przy zachowaniu intuicyjnej obsługi interfejsu,
- realizowanie analizy statycznej plików źródłowych użytkownika oraz graficzne oznaczanie wykrytych odwołań do przestarzałego oprogramowania,
- udostępnianie menu kontekstowego pozwalającego użytkownikowi na podjęcie decyzji o wyborze sztywnej podmiany regułowej lub transformacji z wykorzystaniem sztucznej inteligencji,
- wizualizowanie proponowanych zmian w oknie różnicowym przed dokonaniem jakichkolwiek trwałych modyfikacji w strukturze plików,
- obsługiwanie stabilnej komunikacji z silnikami analizy kodu oraz modelami sztucznej inteligencji.

3.2.2. Niezależny moduł konsolowy

Moduł wiersza poleceń służy do automatyzowania weryfikacji jakości kodu na poziomie serwerów budujących w trybie bezinterfejsowym. Dla komponentu tego definiuje się następujące wymagania:

- prosty sposób inicjalizacji z poziomu terminala systemowego,
- zdolność do samodzielnego interpretowania zależności projektu bez konieczności uruchamiania środowiska graficznego,
- generowanie czytelnych raportów wyjściowych dotyczących użycia przestarzałych elementów w skanowanych projektach.

3.2.3. Silnik ekstrakcji metadanych i transformacji kodu

Silnik odpowiada za mechanizm pozyskiwania wiedzy z pakietów zewnętrznych oraz realizację samej refaktoryzacji. Oczekuje się, że komponent ten sprosta następującym wymaganiom:

- automatyczne skanowanie kodu zewnętrznych bibliotek i bezbłędne wyciąganie informacji o wycofywanych strukturach wprost z metadanych dokumentacji Javadoc,
- translacja odnalezionych reguł tekstowych na gotowe instrukcje modyfikacji kodu deweloperskiego,
- obsługa akcji naprawczych obejmujących bezpośrednią podmianę przestarzałego wywołania, wstawienie komentarza z planowanym zadaniem deweloperskim oraz dodanie odpowiedniej adnotacji wyciszającej ostrzeżenia kompilatora,
- przekształcanie lokalnego kontekstu programu na zoptymalizowane zapytania i przysyłanie ich do modeli językowych w przypadku wyboru inteligentnej ścieżki naprawczej AI.

4. Technologie i narzędzia wykorzystane w rozwiązaniu

Niniejszy rozdział prezentuje architekturę narzędziową zrealizowanego prototypu. W pierwszej kolejności omówiono użyty język programowania oraz platformę bazową oraz specyfikę tworzenia rozszerzeń w ekosystemie IntelliJ Platform SDK. Następnie przedstawiono sposób integracji dużych modeli językowych, opisując wykorzystanie chmurowego interfejsu oraz lokalnych modeli. Rozdział zamyka przegląd infrastruktury budowania projektu.

4.1. Java

Środowiskiem docelowym dla zrealizowanego prototypu jest platforma IntelliJ [13], dlatego jako główny język programowania wybrano język Java. Decyzja ta podyktowana była pełną kompatybilnością ze środowiskiem uruchomieniowym Java Virtual Machine [17] oraz natywnym wsparciem ze strony oficjalnego pakietu narzędziowego IntelliJ Platform SDK [14].

Java to uniwersalny, obiektowy język programowania wysokiego poziomu, stworzony przez firmę Sun Microsystems, a obecnie rozwijany przez Oracle Corporation, który od połowy lat 90. XX wieku stanowi jeden z fundamentów światowej inżynierii oprogramowania. Architektura języka została zaprojektowana wokół zasady WORA (*Write Once, Run Anywhere*), co oznacza, że kod źródłowy kompilowany jest do niezależnego od architektury sprzętowej kodu bajtowego (ang. *bytecode*). Wykonanie tego kodu odbywa się następnie za pośrednictwem Wirtualnej Maszyny Javy (ang. *Java Virtual Machine*), co gwarantuje wysoką przenośność aplikacji pomiędzy różnymi systemami operacyjnymi.

Kluczową cechą Javy jest jej silne, statyczne typowanie. Wymusza ono na programiście deklarację typów zmiennych już na etapie pisania kodu, co pozwala kompilatorowi na wczesne wykrywanie błędów semantycznych i strukturalnych. Przekłada się to bezpośrednio na wyższą stabilność i bezpieczeństwo tworzonych systemów, co jest szczególnie istotne w przypadku rozbudowanych aplikacji korporacyjnych.

Oprócz samej składni, o sile Javy decyduje jej ekosystem. Składają się na niego nie tylko bogata biblioteka standardowa, ale również dojrzałe systemy budowania np. Maven, Gradle, zaawansowane środowiska programistyczne np. IntelliJ IDEA, Eclipse oraz ogromna baza rozwiązań otwarto-źródłowych (ang. *open-source*), które znacząco przyspieszają proces wytwarzania nowoczesnego oprogramowania.

4.2. Ekosystem IntelliJ Platform SDK

Realizacja narzędzia do analizy i modyfikacji kodu wymagała głębokiej integracji ze środowiskiem programistycznym. Wybór platformy IntelliJ, rozwijanej przez firmę JetBrains, podyktowany był jej dominującą pozycją na rynku narzędzi dla języka Java oraz niezwykle rozbudowanym interfejsem programowania aplikacji dla twórców rozszerzeń - IntelliJ Platform SDK.

4.2.1. Architektura rozszerzeń i cykl życia wtyczki

Architektura wtyczek w środowisku IntelliJ opiera się na deklaratywnym modelu konfiguracji. Głównym punktem wejścia i manifestem każdego rozszerzenia jest plik `plugin.xml`. Definiuje on metadane

wtyczki, jej zależności od bazowych modułów platformy oraz rejestruje kluczowe komponenty systemu, poprzez które wtyczka integruje się z cyklem życia środowiska IDE.

Tworzenie rozszerzeń w ekosystemie IntelliJ Platform SDK oferuje szeroki wachlarz wzorców architektonicznych. W kontekście narzędzi analitycznych i asystentów programowania, najistotniejsze z nich to:

- **Okna narzędziowe (ang. *Tool Windows*):** Stanowią główne punkty interakcji z użytkownikiem dla rozszerzeń. Są to dedykowane panele, które pozwalają na osadzenie interfejsu graficznego w oknie edytora kodu.
- **Ustawienia i konfiguracja (ang. *Configurables*):** Ekosystem dostarcza specjalne punkty rozszerzeń (ang. *Extension Points*), pozwalające na wstrzykiwanie własnych paneli konfiguracyjnych bezpośrednio do głównego okna ustawień środowiska.
- **Serwisy (ang. *Services*):** Komponenty odpowiedzialne za przechowywanie stanu aplikacji oraz realizację logiki biznesowej odseparowanej od warstwy prezentacji. Serwisy mogą być rejestrowane na poziomie całej aplikacji (ang. *Application-level*) lub pojedynczego projektu (ang. *Project-level*). Zgodnie z rygorystycznymi wytycznymi optymalizacyjnymi środowiska, serwisy te inicjowane są leniwie (ang. *lazy loading*), co gwarantuje, że nie wydłużają one czasu uruchamiania samego środowiska IDE.

Uproszczoną, teoretyczną strukturę deklaracji tych komponentów w pliku konfiguracyjnym przedstawiono na Listingu 4.1.

Listing 4.1: Przykładowy plik plugin.xml rejestrujący okno narzędziowe. Źródło: opracowanie własne.

```
1 <idea-plugin>
2   <depends>com.intellij.modules.platform</depends>
3
4   <extensions defaultExtensionNs="com.intellij">
5     <toolWindow id="ExampleToolWindow"
6               anchor="right"
7               factoryClass="com.example.ui.MyToolWindowFactory"/>
8
9     <applicationConfigurable
10      parentId="tools"
11      instance="com.example.settings.MyConfigurable"
12      id="com.example.settings.MyConfigurable"
13      displayName="Plugin Settings"/>
14
15     <applicationService
16      serviceImplementation="com.example.services.MySettingsService"/>
17   </extensions>
18 </idea-plugin>
```

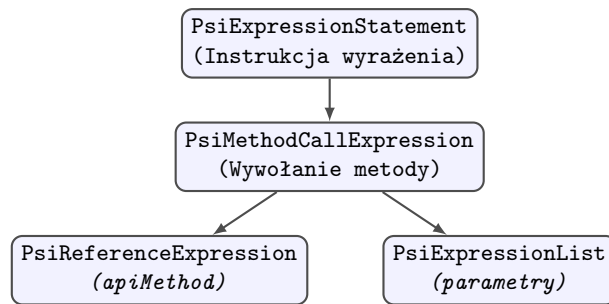
4.2.2. Model Reprezentacji Kodu

Fundamentem każdego narzędzia diagnostycznego działającego w ekosystemie IntelliJ jest *Program Structure Interface* (PSI). Jest to wewnętrzny, wysoce wyspecjalizowany silnik platformy, który analizuje kod źródłowy i przekształca go w wielopoziomowe, semantyczne drzewo składniowe.

W przeciwieństwie do klasycznych, płaskich drzew AST (ang. *Abstract Syntax Tree*), węzły drzewa PSI - reprezentowane przez obiekty `PsiElement` - posiadają bogaty kontekst wewnętrznyjęzykowy.

Uproszczoną strukturę hierarchiczną takiego drzewa dla pojedynczego wywołania metody zaprezentowano na Rysunku 4.1.

Architektura PSI dostarcza twórcom rozszerzeń szereg zaawansowanych mechanizmów analitycznych i manipulacyjnych, wśród których do najważniejszych w kontekście refaktoryzacji należą:



Rysunek 4.1: Uproszczona struktura drzewa PSI dla wywołania metody. Źródło: opracowanie własne.

1. **Precyzyjna detekcja i nawigacja:** Dzięki klasom, takim jak `PsiMethodCallExpression` czy `PsiAnnotation`, API pozwala na deterministyczne lokalizowanie konkretnych konstrukcji językowych. Mechanizm ten opiera się na analizie semantycznej, co eliminuje problemy związane ze zwykłym wyszukiwaniem tekstowym, ignorując m.in. zbieżności nazw w komentarzach czy łańcuchach znaków.
2. **Ekstrakcja semantycznego kontekstu:** Drzewo PSI daje pełen dostęp do metadanych analizowanego kodu. Możliwe jest m.in. odczytanie sygnatury metody, rozwiązywanie typów zmiennych oraz badanie struktury hierarchicznej klas. Stanowi to ustrukturyzowane źródło wiedzy, niezbędne do zasilania narzędzi zewnętrznych czy wstrzykiwania kontekstu do analizy heurystycznej.
3. **Bezpieczna modyfikacja kodu:** Transformacje kodu źródłowego za pomocą PSI nie odbywają się poprzez ryzykowne operacje na surowym tekście. Zamiast tego platforma dostarcza narzędzia np. `PsiElementFactory`, które służą do programatycznego generowania nowych poddrzew, a następnie bezpiecznego podmieniania całych węzłów. Środowisko automatycznie dba o spójność składniową, reformatowanie kodu oraz aktualizację referencji (ang. *imports*), redukując ryzyko powstania błędów kompilacji.

4.3. Wsparcie dużych modeli językowych

Współczesne podejścia do automatyzacji refaktoryzacji kodu coraz częściej opierają się na wykorzystaniu dużych modeli językowych (ang. *Large Language Models*, LLM). Zastosowanie sztucznej inteligencji pozwala na wyjście poza sztywne, oparte na regułach transformatory i umożliwia analizę semantyczną oraz kontekstową. Ze względu na zróżnicowane wymagania dotyczące wydajności, kosztów oraz prywatności danych, ekosystem narzędziowy w tym obszarze dzieli się na rozwiązania chmurowe oraz modele uruchamiane lokalnie.

4.3.1. Modele chmurowe

Modele przetwarzane w chmurze oferują najwyższą moc obliczeniową oraz zdolność analizy bardzo szerokiego kontekstu architektonicznego. Do wiodących rozwiązań w tej kategorii, powszechnie wykorzystywanych w zaawansowanych narzędziach inżynierii oprogramowania, należą między innymi:

- **Rodzina modeli Gemini (Google):** Wyróżniająca się natywną multimodalnością oraz pojemnym oknem kontekstowym, co umożliwia jednoczesną analizę zawartości obszernych repozytoriów oraz śledzenie głębokich zależności pomiędzy wieloma modułami projektu [11].
- **Rodzina modeli GPT (OpenAI):** Obejmująca zaawansowane modele, które wyznaczają rynkowe standardy w zakresie wnioskowania logicznego i generowania kodu źródłowego [25].

- **Rodzina modeli Claude (Anthropic):** Modele te charakteryzują się wysoką precyzją w zadaniach związanych z parsowaniem i refaktoryzacją skomplikowanych struktur kodu, kładąc jednocześnie silny nacisk na bezpieczeństwo i przewidywalność generowanych odpowiedzi [1].

Komunikacja z chmurowymi modelami LLM realizowana jest zazwyczaj synchronicznie za pośrednictwem dedykowanego interfejsu programistycznego - REST API. Wykorzystanie tego podejścia pozwala na generowanie wysoce precyzyjnych transformacji kodu w sytuacjach, gdy migracja wymaga zrozumienia nie tylko pojedynczej metody, ale również złożonego grafu wywołań w całym projekcie. Głównym ograniczeniem architektury chmurowej jest jednak wymóg stałego połączenia z siecią, a także kwestie prywatności. Przetwarzanie fragmentów bazy kodowej na serwerach zewnętrznego dostawcy usług nierzadko stoi w sprzeczności z politykami korporacyjnymi, które zakazują wyprowadzania własności intelektualnej poza lokalną infrastrukturę firmy.

4.3.2. Lokalne modele generatywne i środowisko uruchomieniowe

W odpowiedzi na wymagania dotyczące prywatności i bezpieczeństwa kodu źródłowego, dynamicznie rozwija się nurt modeli uruchamianych w pełni lokalnie. Ogromną rolę w tym procesie odgrywa społeczność skupiona wokół platformy *Hugging Face* [12], która stanowi centralne repozytorium modeli otwartoźródłowych.

Uruchomienie takich modeli wymaga odpowiedniego oprogramowania. Choć istnieją niskopoziomowe silniki, jak *llama.cpp* [9], czy platformy graficzne, np. *LM Studio* [18], wygodną warstwę abstrakcji nad modelami stanowi często oprogramowanie takie jak *Ollama* [24]. Pełni ono rolę lokalnego serwera, udostępniając spójne API do komunikacji z modelami, eliminując konieczność ręcznej konfiguracji środowisk hostujących.

Wśród lokalnych modeli, na szczególną uwagę zasługują:

- modele z rodziny Gemma [8] opracowane przez firmę Google,
- modele Qwen [2] stworzone przez firmę Alibaba Cloud,
- modele Llama [21] stworzone przez firmę Meta.

Pomimo mniejszej liczby parametrów w porównaniu do flagowych rozwiązań chmurowych, wykazują one wysoką skuteczność w specyficznych zadaniach inżynierskich, przy jednoczesnym zagwarantowaniu, że analizowany kod źródłowy nie opuszcza maszyny programisty.

Efektywność tych otwartoźródłowych rozwiązań w zadaniach przekształcania kodu została jednak poddana weryfikacji w benchmarku SWE-Refactor z lutego 2026 roku [36]. Wyniki eksperymentów pokazały, że mniejsze, modele, takie jak Qwen czy Llama, radzą sobie poprawnie niemal wyłącznie z prostymi, lokalnymi edycjami o charakterze atomowym, skupiając się głównie na wydzielaniu pojedynczych metod. W przypadku napotkania złożonych transformacji wieloplikowych lub struktur silnie zależnych ich samodzielna skuteczność spada niemal do zera. Jednoznacznie dowodzi to, że lokalne środowisko uruchomieniowe wymaga ścisłego wsparcia ze strony systemu analizy statycznej środowiska IDE w celu zapewnienia spójności i poprawności semantycznej całego repozytorium.

4.3.3. Inżynieria zapytań

Skuteczność dużych modeli językowych jest silnie uzależniona od techniki formułowania instrukcji, określanej mianem inżynierii zapytań (ang. *prompt engineering*). Aby precyzyjnie poinstruować model i zminimalizować ryzyko tzw. halucynacji, konieczne jest dostarczenie mu nie tylko samego polecenia, ale również bogatego, wyodrębnionego kontekstu, np. specyfikacji, fragmentów dokumentacji czy struktury analizowanego kodu.

Komunikacja z nowoczesnymi modelami najczęściej opiera się na ustandaryzowanym formacie zapytań. Format ten dzieli przekazywane informacje na role:

- **system:** definiuje nadrzędne zachowanie agenta i wymusza pożądany format wyjściowy,

- **user**: dostarcza właściwe zadanie wraz z wyekstrahowanym kontekstem.

Poza samą treścią zapytań, kluczową rolę w sterowaniu zachowaniem modelu odgrywają parametry konfiguracyjne procesu generowania, tzw. hiperparametry. W przypadku ustrukturyzowanej komunikacji z interfejsem API najważniejsze z nich to:

- **temperature** (temperatura) – określa stopień losowości w dobieraniu kolejnych tokenów. W zadaniach związanych z automatyczną refaktoryzacją kodu, gdzie priorytetem jest determinizm i poprawność składniowa, zaleca się stosowanie wartości bliskich zeru. Zapobiega to nadmiernej kreatywności modelu.
- **top_p** (próbkiowanie jądrowe) – działa komplementarnie do temperatury, ograniczając pulę rozważanych tokenów wyłącznie do tych, których skumulowane prawdopodobieństwo przekracza zadaną wartość *p*.
- **max_tokens** (limit tokenów) – zabezpiecza system przed generowaniem zbyt długich, niepożądanych odpowiedzi. Stanowi również główny mechanizm kontroli kosztów odpytywania płatnych interfejsów.
- **stop** (sekwencje zatrzymujące) – definiuje ciągi znaków np. pojedynczy znak `}` lub znaczniki bloku `““`, po wygenerowaniu których model natychmiast przerywa działanie. Jest to kluczowy bezpiecznik zapobiegający dopisywaniu nadmiarowego kodu.

Uproszczony, teoretyczny schemat takiego zautomatyzowanego zapytania, uwzględniający wybrane hiperparametry, zaprezentowano na Listingu 4.2.

Listing 4.2: Teoretyczny schemat ustrukturyzowanego zapytania API do modelu językowego wraz z hiperparametrami. Źródło: opracowanie własne.

```

1 {
2   "model": "przykladowy-model-llm",
3   "temperature": 0.0,
4   "top_p": 1.0,
5   "max_tokens": 1024,
6   "stop": ["}", "““"],
7   "messages": [
8     {
9       "role": "system",
10      "content": "Jestes asystentem inzynierii oprogramowania. Zwracaj
        wylacznie poprawny kod Java, bez wyjasnien."
11    },
12    {
13      "role": "user",
14      "content": "Zoptymalizuj ponizsza funkcje: \n\n public void oldMethod
        () { ... }"
15    }
16  ]
17 }
```

4.4. System budowania Gradle

Wytwarzanie rozszerzeń dla platformy IntelliJ wymaga specyficznego i ustandaryzowanego podejścia do procesu kompilacji, rozwiązywania zależności oraz pakowania kodu. W ekosystemie języka Java

funkcję tę pełni Gradle - zaawansowany system automatyzacji budowania projektów, rekomendowany przez firmę JetBrains.

Kluczowym elementem wspierającym ten proces jest rozszerzenie `intellij-platform-gradle-plugin`. Stanowi ono warstwę abstrakcji, która automatyzuje większość rutynowych zadań, w tym:

- Pobieranie i konfigurację instancji docelowego środowiska IntelliJ, względem którego kompilowany i testowany jest kod źródłowy.
- Walidację kompatybilności binarnej wtyczki za pomocą narzędzia *Plugin Verifier*, co zapobiega błędom w środowisku produkcyjnym.
- Generowanie ostatecznego, gotowego do instalacji archiwum dystrybucyjnego.

Dzięki zastosowaniu systemu Gradle proces budowania rozszerzenia staje się w pełni powtarzalny i niezależny od lokalnej konfiguracji stacji roboczej programisty.

5. Implementacja wtyczki do IntelliJ IDEA wspomagającej zarządzanie bibliotekami zewnętrznymi

W niniejszym rozdziale szczegółowo opisano techniczne aspekty realizacji prototypu systemu wspomagającego zarządzanie bibliotekami zewnętrznymi. Treść tej części pracy koncentruje się na omówieniu architektury systemu, przyjętego schematu działania algorytmu oraz zaprojektowanym modelu komunikacji.

5.1. Architektura systemu i podział na warstwy logiczne

Od strony strukturalnej projekt został zorganizowany wokół szablonu `intellij-platform-plugin-template` i oparty na systemie budowania Gradle. Zgodnie z dobrymi praktykami inżynierii oprogramowania (m.in. zasadą *Separation of Concerns*), kod źródłowy podzielono na wyraźne warstwy logiczne, odzwierciedlające przepływ informacji w systemie:

1. **Warstwa integracji z ekosystemem Maven** – moduł ten odpowiada za parsowanie lokalnych plików `pom.xml` oraz komunikację z repozytorium Maven Central w celu weryfikacji dostępnych wersji oraz pobrania archiwów źródłowych. Kluczowymi komponentami tej warstwy są `VersionAnalysisService` oraz `ProjectDependencyResolver`.
2. **Warstwa analizy kodów źródłowych bibliotek zewnętrznych** – zadaniem tego modułu jest budowanie bazy wiedzy o wycofany API. Za pomocą silnika `JavaParser` rekonstruuje on drzewa AST pobranych archiwów `-sources.jar` i wyciąga z nich surowe komentarze Javadoc oraz formalne sugestie migracyjne, mapując je na strukturę `DeprecatedMethodInfo`. Centralnym komponentem jest tutaj klasa `DeprecatedMethodFinder`.
3. **Warstwa skanowania projektu użytkownika przy pomocy modelu PSI** – komponent działający wewnątrz środowiska IntelliJ IDEA. Odpowiada za analizę kodu autorskiego użytkownika bezpośrednio w edytorze. Wykorzystuje semantyczny model *Program Structure Interface* oraz mechanizm wizytatorów do natychmiastowego mapowania wywołań wewnątrz otwartej przestrzeni roboczej, opierając się na klasie `MethodUsageScanner`.
4. **Warstwa skanowania projektu użytkownika przy pomocy biblioteki ASM** – niezależny komponent przeznaczony do weryfikacji produktów końcowych użytkownika w trybie CLI. Wykorzystuje silnik ASM do skanowania fizycznych plików `.class` znajdujących się w katalogach wyjściowych kompilatora, takich jak `target` lub `build`, identyfikując instrukcje wywołań przestarzałych metod na poziomie kodu bajtowego za pomocą klasy `BytecodeDeprecationScanner`.
5. **Warstwa integracji z AI** – komponent zajmujący się inżynierią promptów. Logika tworzenia zapytań w metodzie `buildPrompt()` oraz komunikacja z zewnętrznym API modelu językowego za pomocą metody `callModelAPI()` zostały zamknięte w klasie `AIRefactoringService`.
6. **Warstwa transformacji kodu** - warstwa dostarczająca niemutowalne struktury danych takie jak `MethodUsageInfo` czy `DeprecatedMethodInfo` oraz silniki modyfikacji plików. Warstwa ta

zarządza nakładaniem poprawek na drzewo PSI przy użyciu operacji `WriteCommandAction` oraz bezpośrednimi modyfikacjami wirtualnych dokumentów tekstowych.

7. **Warstwa prezentacji i interfejsu użytkownika** – moduł zawierający komponenty graficzne osadzone natywnie w oknach narzędziowych IDE, takie jak `JPanel` czy `JTable`, udostępniające wyniki analizy użytkownikowi. Komponent ten odpowiada również za bezpośrednie przechwytywanie akcji użytkownika, zarządzanie oknami dialogowymi podglądu różnicowego edytora oraz synchronizację zdarzeń graficznych z asynchronicznymi wątkami tła aplikacji.

Taki podział zapewnia wysoką rozszerzalność narzędzia i pozwala na niezależny rozwój poszczególnych modułów. Szczegółowa specyfikacja techniczna, implementacja oraz wzorce projektowe zastosowane w obrębie każdej z wyżej wymienionych warstw zostały szczegółowo omówione w kolejnych podrozdziałach niniejszej pracy.

5.2. Integracji z ekosystemem Maven

Autonomiczne działanie systemu w zakresie wykrywania przestarzałych bibliotek oraz analizy ryzyka związanego z ich aktualizacją wymaga ścisłej integracji z ekosystemem narzędzia Maven. Proces ten realizowany jest dwuetapowo, począwszy od identyfikacji lokalnych zależności zadeklarowanych w projekcie użytkownika, aż po komunikację ze zdalnym repozytorium Maven Central w celu pobrania metadanych i archiwów źródłowych.

Za parsowanie lokalnych plików konfiguracyjnych `pom.xml` odpowiada dedykowany mechanizm zaimplementowany w klasie `MethodUsageScanner`. Sposób, w jaki system przetwarza strukturę pliku bez konieczności ręcznego parsowania surowego formatu XML, zaprezentowano na Listingu 5.1.

Listing 5.1: Odczyt i parsowanie zależności z pliku `pom.xml` przy użyciu `MavenXpp3Reader`. Źródło: opracowanie własne.

```
1 private List<Dependency> readDependenciesFromPom(VirtualFile pomFile) throws
   Exception {
2     List<Dependency> dependencies = new ArrayList<>();
3     MavenXpp3Reader reader = new MavenXpp3Reader();
4
5     try (InputStream inputStream = pomFile.getInputStream()) {
6         org.apache.maven.model.Model model = reader.read(new InputStreamReader(
7             inputStream));
8         for (org.apache.maven.model.Dependency dep : model.getDependencies()) {
9             dependencies.add(new Dependency(dep.getGroupId(), dep.getArtifactId(),
10                dep.getVersion()));
11         }
12     }
13     return dependencies;
14 }
```

Wykorzystanie oficjalnego parsera `MavenXpp3Reader` pozwala na zmapowanie pliku XML bezpośrednio na obiektowy model struktury projektu. Wtyczka wyodrębnia w ten sposób unikalne koordynaty każdej biblioteki takie jak `groupId`, `artifactId` oraz aktualnie używaną wersję `version`, pakując je w rekordy ułatwiające dalsze przetwarzanie.

Głównym zadaniem integracji ze zdalnym repozytorium Maven Central jest pozyskiwanie zasobów niezbędnych do identyfikacji zamienników dla przestarzałego kodu w ramach bieżącej konfiguracji projektu użytkownika. Do przeprowadzenia analizy statycznej klasa `DeprecatedMethodFinder` konstruuje ścieżkę sieciową do oficjalnego archiwum z kodem źródłowym analizowanej biblioteki. Repozytorium Maven Central nie udostępnia klasycznego interfejsu REST API, a cała wymiana danych opiera się na deterministycznej konwencji układu repozytorium znanej pod nazwą *Maven Repository Layout*. System

konstruuje przewidywalne adresy URL i pobiera zasoby za pomocą standardowych żądań HTTP GET, co nie wymaga uwierzytelniania ani stosowania kluczy dostępowych.

Punktem wejścia do zdalnego repozytorium, zapisanym w kodzie jako stała `MAVEN_CENTRAL_BASE`, jest oficjalny adres głównego serwera Maven <https://repo1.maven.org/maven2>. Kluczowym zabiegiem przy budowaniu zapytania o pakiet źródłowy jest translacja identyfikatora grupy na odpowiednią ścieżkę sieciową oraz precyzyjne wskazanie aktualnie zadeklarowanej wersji (zob. Listing 5.2).

Listing 5.2: Konstrukcja adresu URL do pobrania archiwum z kodem źródłowym. Źródło: opracowanie własne.

```
1 String jarUrl = String.format("%s/%s/%s/%s/%s",
2     MAVEN_CENTRAL_BASE,
3     groupId.replace('.', '/'), //np. "com.google.code.gson" -> "com/google/code/gson"
4     artifactId,
5     version,
6     artifactId + "-" + version + "-sources.jar"
7 );
```

Zamiana kropek na separatory ścieżki za pomocą metody `replace('.', '/')` pozwala na poprawne odpytanie serwera zgodnie z wytycznymi konwencji Maven. Obecność specyficznego sufiksu `-sources.jar` oznaczającego źródła jest standardowym klasyfikatorem w ekosystemie Maven i wskazuje na artefakt zawierający nie tylko skompilowany kod bajtowy, ale również oryginalne pliki źródłowe java wraz z kompletnymi komentarzami dokumentacji Javadoc. Pobranie i rozpakowanie tego archiwum w lokalnej pamięci otwiera drogę do uruchomienia dedykowanych parserów abstrakcyjnego drzewa składniowego, co zostało szczegółowo opisane w podrozdziale 5.3.

Drugą funkcjonalnością systemu, która wprost rozszerza proces podstawowej detekcji o proaktywne zarządzanie długiem technologicznym w projekcie, jest moduł wyszukiwania wyższych wydań zależności oraz sugerowania bezpiecznych migracji realizowany przez klasę `VersionAnalysisService`. Proces ten opiera się na dynamicznym odpytywaniu zdalnego repozytorium o aktualne metadane, co pozwala na precyzyjne wyznaczenie ścieżki rozwoju danego komponentu. W celu weryfikacji dostępnych wydań oprogramowania wtyczka buduje zapytanie o plik konfiguracyjny, którego dynamiczną strukturę adresu URL przedstawia Listing 5.3.

Listing 5.3: Dynamiczna konstrukcja adresu URL dla metadanych biblioteki. Źródło: opracowanie własne.

```
1 String metadataUrl = String.format("%s/%s/%s/maven-metadata.xml",
2     MAVEN_CENTRAL_BASE,
3     groupId.replace('.', '/'),
4     artifactId
5 );
```

Pobrany w odpowiedzi dokument XML posiada ściśle zdefiniowaną strukturę, której wycinek zaprezentowano na Listingu 5.4.

Listing 5.4: Przykładowa struktura odpowiedzi `maven-metadata.xml` pobranej z repozytorium. Źródło: opracowanie własne.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <metadata>
3   <groupId>com.google.code.gson</groupId>
4   <artifactId>gson</artifactId>
5   <versioning>
6     <latest>2.10.1</latest>
7     <release>2.10.1</release>
8     <versions>
9       <version>2.8.9</version>
```

```

10     <version>2.9.0</version>
11     <version>2.9.1</version>
12     <version>2.10</version>
13     <version>2.10.1</version>
14 </versions>
15 <lastUpdated>20230106000000</lastUpdated>
16 </versioning>
17 </metadata>

```

Jak ukazuje powyższy przykład, otrzymana odpowiedź zawiera sekcję `<versioning>`, z której metoda `getAvailableVersions` wyciąga pełną historię wydań artefaktu. Algorytm wyszukuje wszystkie węzły oznaczające konkretną wersję za pomocą wskazania nazwy tagu, a następnie sortuje je chronologicznie oraz logicznie przy użyciu klasy `ComparableVersion` pochodzącej z pakietu narzędziowego Maven.

Po wyznaczeniu poprawnie posortowanej listy wydań klasa `VersionAnalysisService` uruchamia procedurę porównawczą zaimplementowaną w metodzie `compareVersions`. Mechanizm ten wykorzystuje pobrane wcześniej przez skaner archiwum wersji bieżącej i za pomocą opisanego wyżej protokołu pobiera z repozytorium wyłącznie jedno dodatkowe archiwum `sources jar` odpowiadające najnowszemu stabilnemu wydaniu zidentyfikowanemu w metadanych jako wersja docelowa. Analizator dekoduje i bada oba strumienie danych w celu precyzyjnego wykrycia, które metody zostały oznaczone jako przestarzałe oraz które zostały całkowicie usunięte w najnowszych wydaniach. Na podstawie przeprowadzonego porównania system buduje końcowy obiekt klasy `UpgradeRecommendation`, który agreguje kompletne, tekstowe rekomendacje migracyjne dla programisty prezentowane bezpośrednio w interfejsie graficznym lub przestrzeni CLI.

5.3. Analiza kodów źródłowych bibliotek zewnętrznych

Po pobraniu archiwów źródłowych z repozytorium Maven, system przystępuje do fazy budowania lokalnej bazy wiedzy o przestarzałym API. Zadanie to realizowane jest przez klasę `DeprecatedMethodFinder`. Konieczność implementacji tego modułu wynika bezpośrednio z ograniczeń kompilatorowych języka Java - informacje o adnotacjach są zachowywane w bytcode, jednak opisy tekstowe, powody deprecjacji oraz intencje autorów udokumentowane w tagu `@deprecated` są bezpowrotnie usuwane podczas budowania plików `.class`.

W celu odzyskania tych danych wtyczka wypakowuje pobrany plik archiwum do katalogu tymczasowego, a następnie uruchamia silnik `JavaParser`. Ponieważ transformacja składniowa realizowana jest indywidualnie dla każdej jednostki kompilacji, system wymagał wdrożenia warstwy pośredniej sterującej dostarczaniem plików wejściowych. Zadanie to powierzono mechanizmowi skanującemu, który mapuje zawartość pamięci tymczasowej i koordynuje przepływ danych do parsera. Algorytm odpowiedzialny za przeszukiwanie struktury katalogów tymczasowych oraz główną orkiestrację procesu analizy pojedynczego pliku zaprezentowano na Listingu 5.5.

Listing 5.5: Strumieniowe przeszukiwanie katalogu oraz orkiestracja analizy pliku źródłowego. Źródło: opracowanie własne.

```

1 private void analyzeFiles(Path dir, Dependency dependency, List<DeprecatedMethodInfo>
  output) throws IOException {
2     if (!Files.exists(dir)) return;
3     Files.walk(dir)
4         .filter(path -> path.toString().endsWith(".java"))
5         .forEach(path -> {
6             try {
7                 analyzeFile(path.toFile(), dependency, output);
8             } catch (Exception e) {
9                 LOGGER.error("Error analyzing file {}", path, e);

```

```

10         }
11     });
12 }
13
14 private void analyzeFile(File file, Dependency dependency, List<DeprecatedMethodInfo>
    output) throws IOException {
15     CompilationUnit cu = StaticJavaParser.parse(file);
16
17     cu.findAll(MethodDeclaration.class).forEach(method -> {
18         if (!isDeprecated(method)) {
19             return;
20         }
21
22         String className = getClassName(cu, method);
23         String methodName = method.getNameAsString();
24
25         if (isMethodIgnored(className, methodName)) {
26             LOGGER.debug("Method {}:{} is ignored, skipping", className, methodName);
27             return;
28         }
29
30         String deprecationReason = extractDeprecationReason(method);
31         List<String> suggestedReplacements = extractSuggestedReplacements(method, cu)
32             ;
33         DeprecatedMethodInfo methodInfo = buildMethodInfo(file, dependency, method,
34             className, deprecationReason, suggestedReplacements);
35
36         output.add(methodInfo);
37     });
38 }
39
40 private boolean isDeprecated(MethodDeclaration method) {
41     return method.getAnnotations().stream().anyMatch(annotation ->
42         annotation.getNameAsString().equals("Deprecated"));
43 }

```

W przedstawionym fragmencie kodu metoda `analyzeFiles` wykorzystuje klasę `Files` do rekurencyjnego przejścia przez strukturę drzewa katalogów wypakowanej biblioteki. Każdy zidentyfikowany plik trafia do metody `analyzeFile`, gdzie klasa `StaticJavaParser` przetwarza wejście i generuje strukturę abstrakcyjnego drzewa składniowego odzwierciedloną przez klasę `CompilationUnit`.

Zaimplementowany algorytm opiera się na podejściu deklaratywnym. Program odpytuje strukturę pod kątem wszystkich węzłów reprezentujących deklaracje metod za pomocą klasy `MethodDeclaration`, a następnie odrzuca elementy niespełniające kryteriów za pomocą metody `isDeprecated`. Po weryfikacji unikalności oraz sprawdzeniu konfiguracji ignorowanych komponentów, system przechodzi do wykonania procedur odpowiedzialnych za ekstrakcję komentarzy, co przedstawia Listing 5.6.

Listing 5.6: Algorytmy ekstrakcji uzasadnienia deprecjacji oraz rekomendowanych zamienników API.
Źródło: opracowanie własne.

```

1 private String extractDeprecationReason(MethodDeclaration method) {
2     if (method.getJavadoc().isEmpty()) {
3         return "";
4     }
5
6     var deprecatedTag = method.getJavadoc().get().getBlockTags().stream()
7         .filter(tag -> tag.getType() == JavadocBlockTag.Type.DEPRECATED)
8         .findFirst();
9
10    if (deprecatedTag.isEmpty()) {
11        return "";
12    }

```

```

13
14     String content = deprecatedTag.get().getContent().toText();
15     if (content.contains("Use") || content.contains("use")) {
16         int useIndex = content.toLowerCase().indexOf("use");
17         if (useIndex > 0) {
18             return content.substring(0, useIndex).trim();
19         }
20     }
21     return "";
22 }
23
24 private List<String> extractSuggestedReplacements(MethodDeclaration method,
25     CompilationUnit cu) {
26     List<String> suggestedReplacements = new ArrayList<>();
27
28     if (method.getJavadoc().isPresent()) {
29         var deprecatedTag = method.getJavadoc().get().getBlockTags().stream()
30             .filter(tag -> tag.getType() == JavadocBlockTag.Type.DEPRECATED)
31             .findFirst();
32
33         if (deprecatedTag.isPresent()) {
34             deprecatedTag.get().getContent().getElements().stream()
35                 .filter(e -> e instanceof JavadocInlineTag)
36                 .map(e -> (JavadocInlineTag) e)
37                 .filter(inlineTag -> inlineTag.getType() == JavadocInlineTag.
38                     Type.LINK)
39                 .forEach(linkTag -> {
40                     String link = linkTag.getContent().trim();
41                     suggestedReplacements.add(link.startsWith("#") ? link.
42                         substring(1) : link);
43                 });
44         }
45     }
46
47     if (suggestedReplacements.isEmpty()) {
48         String methodName = method.getNameAsString();
49         cu.findAll(MethodDeclaration.class).stream()
50             .filter(candidate -> !candidate.equals(method))
51             .filter(candidate -> candidate.getNameAsString().contains(methodName) ||
52                 methodName.contains(candidate.getNameAsString()))
53             .map(NodeWithSimpleName::getNameAsString)
54             .forEach(suggestedReplacements::add);
55     }
56     return suggestedReplacements;
57 }

```

Metoda `extractDeprecationReason` bada obecność komentarza i wyodrębnia treść tagu blokowego reprezentowanego przez klasę `JavadocBlockTag`. Tekst poddawany jest analizie pod kątem wystąpienia fraz kluczowych w celu odseparowania właściwego opisu od bezpośrednich instrukcji użycia nowego kodu.

Z kolei metoda `extractSuggestedReplacements` odpowiada za odnalezienie formalnych zamienników. W pierwszej kolejności przeszukuje ona elementy klasy `JavadocInlineTag` o typie ustawionym na znacznik linku, usuwając ewentualne wewnętrzne symbole referencji języka Java. Istotnym elementem algorytmu jest wbudowany mechanizm wyszukiwania zamienników, który uruchamia się automatycznie w przypadku braku jawnych znaczników w dokumentacji. System wykonuje wtedy ponowne zapytanie do struktury `CompilationUnit` i poszukuje alternatywnych metod w obrębie tej samej klasy, wykorzystując kryterium podobieństwa nazewniczego.

Wszystkie zgromadzone metadane są w ostatecznej fazie scalane przez metodę `buildMethodInfo` i pakowane w obiekty modelu `DeprecatedMethodInfo`, tworząc spójny i niezmienny rejestr długu technologicznego dla kolejnych warstw aplikacji.

5.4. Skanowania projektu użytkownika przy pomocy modelu PSI

Mając do dyspozycji kompletną bazę wiedzy o wycofanych komponentach bibliotecznych, system może przystąpić do ich lokalizowania w kodzie napisanym przez programistę. Proces ten realizowany jest w sposób interaktywny wewnątrz edytora IDE, a główny ciężar analityczny spoczywa na klasie `MethodUsageScanner`.

Fundamentem analizy kodu w środowisku IntelliJ IDEA jest model PSI, czyli struktura odwierciedlająca budowę programu w formie drzewa składniowego. Każdy plik źródłowy w projekcie jest tłumaczony na logiczny układ elementów, które są zagnieżdżone jeden w drugim. W schemacie tym znajdują się wszystkie komponenty projektu, począwszy od definicji klas, przez metody i zmienne, aż po pojedyncze znaki oraz słowa kluczowe. Dzięki takiemu podejściu wtyczka ma natychmiastowy dostęp do znaczenia każdego fragmentu, co pozwala na sprawne wyszukiwanie przestarzałych elementów oraz w pełni bezpieczne wprowadzanie poprawek.

Aby efektywnie nawigować po strukturze i precyzyjnie wydobywać interesujące komponenty bez konieczności manualnego implementowania algorytmów przeszukiwania grafów, wtyczka wykorzystuje wzorzec projektowy Wizytator (ang. *Visitor*). Służy do tego dostarczana przez SDK platformy klasa `JavaRecursiveElementVisitor`, która automatycznie i rekursywnie przemierza poszczególne gałęzie drzewa. Nadpisanie jej dedykowanych metod pozwala na filtrowanie i przechwytywanie wyłącznie określonych typów węzłów składniowych, co zaprezentowano na Listingu 5.7.

Listing 5.7: Implementacja wizytatora PSI wykrywającego przestarzałe metody i konstruktory. Źródło: opracowanie własne.

```
1 private List<MethodUsageInfo> scanFileForAllDeprecatedUsages(PsiJavaFile javaFile) {
2     List<MethodUsageInfo> usages = new ArrayList<>();
3
4     javaFile.accept(new JavaRecursiveElementVisitor() {
5         @Override
6         public void visitMethodCallExpression(PsiMethodCallExpression expression) {
7             super.visitMethodCallExpression(expression);
8             PsiMethod resolvedMethod = expression.resolveMethod();
9             if (resolvedMethod != null && isDeprecated(resolvedMethod)) {
10                 DeprecatedMethodInfo info = createDeprecatedMethodInfo(resolvedMethod);
11                 usages.add(createUsageInfo(expression, info, javaFile));
12             }
13         }
14
15         @Override
16         public void visitNewExpression(PsiNewExpression expression) {
17             super.visitNewExpression(expression);
18             PsiMethod constructor = expression.resolveConstructor();
19             if (constructor != null && isDeprecated(constructor)) {
20                 DeprecatedMethodInfo info = createDeprecatedMethodInfo(constructor);
21                 usages.add(createUsageInfoFromNewExpression(expression, info, javaFile));
22             }
23         }
24     });
25     return usages;
26 }
```

Zaimplementowane podejście służy do precyzyjnego lokalizowania miejsc w autorskim kodzie źródłowym użytkownika, w których dochodzi do wywołania metod uznanych za przestarzałe. Zamiast podatnego na błędy, tekstowego przeszukiwania plików, które mogłoby generować fałszywe alarmy w przypadku zbieżności popularnych nazw metod, wtyczka wykorzystuje mechanizm rozwiązywania referencji (ang. *reference resolution*). Poprzez wywołanie metody `expression.resolveMethod()` lub

`expression.resolveConstructor()`, system odpytuje indeksy platformy IntelliJ, aby powiązać konkretną linię kodu napisaną przez programistę z deklaracją metody w zewnętrznej zależności. Pozwala to na bezbłędną identyfikację, czy dane wywołanie faktycznie dotyczy zidentyfikowanego wcześniej przestarzałego API, poprawnie ignorując metody o identycznych nazwach, ale należące do innych, bezpiecznych klas. Gdy wizytator potwierdzi, że dana instrukcja w kodzie użytkownika stanowi element wywołania wycofanego API, uruchamiany jest proces agregacji metadanych na potrzeby przyszłej analizy. Informacje te są pakowane w strukturę `MethodUsageInfo`. Aby wykryty błąd był przydatny dla programisty w trakcie przeglądania wyników, jego statyczna pozycja w drzewie AST, wyrażona jako przesunięcie znakowe (ang. *offset*), musi zostać przełożona na współrzędne pliku. Wtyczka wykorzystuje do tego komponent `PsiDocumentManager`, który tłumaczy indeks węzła na numer linii i kolumny w edytorze tekstowym. Gwarantuje to, że interfejs graficzny wtyczki zaprezentuje użytkownikowi dokładną lokalizację metody i umożliwi natychmiastową nawigację do problematycznego fragmentu kodu.

Ze względu na wielowątkowy charakter IDE, w którym kod może być nieustannie modyfikowany przez użytkownika w trakcie pracy skanera, bezpośredni dostęp do modelu PSI bez dodatkowych zabezpieczeń doprowadziłby do awarii systemu. Z tego względu proces bezpiecznej iteracji po repozytorium został zamknięty wewnątrz natywnej operacji odczytu, co przedstawia Listing 5.8.

Listing 5.8: Proces bezpiecznej iteracji po plikach projektu z wykorzystaniem blokady odczytu. Źródło: opracowanie własne.

```
1 public List<MethodUsageInfo> findAllDeprecatedMethodUsages(boolean fetchDetailedInfo)
2 {
3     return ApplicationManager.getApplication().runReadAction(
4         (Callable<List<MethodUsageInfo>>) () -> {
5         List<MethodUsageInfo> foundUsages = new ArrayList<>();
6         Collection<VirtualFile> javaFiles = FilenameIndex.getAllFilesByExt(
7             project, "java", GlobalSearchScope.projectScope(project)
8         );
9
10        PsiManager psiManager = PsiManager.getInstance(project);
11        ProjectFileIndex fileIndex = ProjectFileIndex.getInstance(project);
12
13        for (VirtualFile file : javaFiles) {
14            if (!fileIndex.isInLibrary(file) && fileIndex.isInSource(file)) {
15                PsiFile psiFile = psiManager.findFile(file);
16                if (psiFile instanceof PsiJavaFile javaFile) {
17                    foundUsages.addAll(scanFileForAllDeprecatedUsages(javaFile));
18                }
19            }
20        }
21
22        return foundUsages;
23    }
24 }
```

Wywołanie metody `runReadAction()` nakłada na czas wykonania przekazanej lambdy blokadę odczytu (ang. *Read Action lock*). Mechanizm ten gwarantuje, że dopóki skaner iteruje po plikach i analizuje drzewo, środowisko zablokuje wszelkie operacje zapisu. Zabezpiecza to proces detekcji przed rzuceniem wyjątku lub analizą niespójnego stanu drzewa programu.

Oprócz zapewnienia bezpieczeństwa współbieżnego, omawiany fragment kodu demonstruje również optymalizację wydajnościową skanera. Zamiast iterować po wszystkich zasobach, wykorzystano klasę `ProjectFileIndex` do odfiltrowania elementów należących do bibliotek zewnętrznych (np. kodu JDK). Ogranicza to przestrzeń poszukiwań wyłącznie do plików źródłowych autorskiego kodu (weryfikacja za pomocą metody `isInSource`), co znacząco skraca czas analizy.

5.5. Skanowanie projektu użytkownika przy pomocy biblioteki ASM

Ostatnim komponentem warstwy detekcji systemu jest silnik analizy skompilowanego kodu bajtowego (ang. *bytecode analysis*), zaimplementowany w klasie `BytecodeDeprecationScanner`. Podczas gdy model PSI opisany w podrozdziale 5.4 operuje w edytorze na plikach źródłowych, silnik oparty na bibliotece ASM realizuje zadanie inspekcji statycznej produktów kompilacji, czyli plików o rozszerzeniu `.class`, znajdujących się w katalogach wyjściowych projektów użytkownika. Pozwala to na masowe i wydajne przetwarzanie całego oprogramowania bez konieczności uruchamiania interfejsu graficznego IDE.

W przeciwieństwie do tradycyjnych parserów budujących pełne drzewa składniowe, biblioteka ASM opiera się na architekturze zorientowanej zdarzeniowo (ang. *Event-driven architecture*). Plik binarny klasy jest sekwencyjnie strumieniowany, a silnik generuje powiadomienia o napotkaniu konkretnych struktur instrukcji. Proces ten został zorganizowany trójwarstwowo. Pierwsza warstwa odpowiada za inicjalizację strumienia danych i konfigurację czytnika kodu bajtowego, co zaprezentowano na Listingu 5.9.

Listing 5.9: Konfiguracja strumienia klasy i wywołanie czytnika `ClassReader`. Źródło: opracowanie własne.

```
1 private List<DeprecatedUsage> analyzeClassFile(Path classFile) throws IOException {
2     List<DeprecatedUsage> usages = new ArrayList<>()
3
4     try (InputStream is = Files.newInputStream(classFile)) {
5         ClassReader classReader = new ClassReader(is);
6         DeprecationDetectorVisitor visitor = new DeprecationDetectorVisitor(usages);
7         classReader.accept(visitor, ClassReader.SKIP_FRAMES);
8     }
9
10    return usages;
11 }
```

Analiza zaprezentowanego fragmentu pozwala na scharakteryzowanie zachowania systemu na poziomie maszyny wirtualnej Java. Komponent `ClassReader` dekoduje surową strukturę bajtów z przekazanego strumienia wejściowego. Kluczowym zabiegiem optymalizacyjnym jest wywołanie metody `accept()` z flagą `ClassReader.SKIP_FRAMES`. Informuje ona silnik, że w trakcie parsowania należy zignorować tablice ramek stosu (ang. *Stack Map Frames*), które są generowane na potrzeby weryfikatora bajtkodu od wersji Java 6. Ponieważ celem wtyczki jest wyłącznie identyfikacja referencji wywołań, pominięcie tych struktur znacząco przyspiesza czas skanowania i redukuje narzut pamięciowy.

Druga warstwa procesu bazuje na podklasie `ClassVisitor`, która mapuje globalny kontekst przetwarzanego pliku i decyduje, które elementy struktury binarnej poddać szczegółowej inspekcji. Implementację tej warstwy przedstawia Listing 5.10.

Listing 5.10: Implementacja klasy `ClassVisitor` śledzącej kontekst i delegującej zadania. Źródło: opracowanie własne.

```
1 private static class DeprecationDetectorVisitor extends ClassVisitor {
2     private final List<DeprecatedUsage> usages;
3     private String currentClassName;
4     private String currentSourceFile;
5
6     public DeprecationDetectorVisitor(List<DeprecatedUsage> usages) {
7         super(Opcode.ASM9);
8         this.usages = usages;
9     }
10
11    @Override
12    public void visit(int version, int access, String name, String signature, String
```

```

13         superName, String[] interfaces) {
14             this.currentClassName = name.replace('/', '.', '.');
15             super.visit(version, access, name, signature, superName, interfaces);
16         }
17
18         @Override
19         public void visitSource(String source, String debug) {
20             this.currentSourceFile = source;
21             super.visitSource(source, debug);
22         }
23
24         @Override
25         public MethodVisitor visitMethod(int access, String name, String descriptor,
26             String signature, String[] exceptions) {
27             return new DeprecationDetectorMethodVisitor(
28                 super.visitMethod(access, name, descriptor, signature, exceptions),
29                 currentClassName, name, currentSourceFile, usages
30             );
31         }
32     }
33 }

```

W konstruktorze omawianego komponentu widoczne jest jawne przekazanie stałej `Opcodes.ASM9` do konstruktora klasy nadrzędnej. Stanowi to mechanizm zabezpieczający stabilność oprogramowania i gwarantujący kompatybilność wsteczną. Informując silnik parsujący o konkretnej edycji specyfikacji biblioteki ASM, dla której zaprojektowano logikę wtyczki, system zabezpiecza się przed awarią w sytuacji, gdy narzędzie zostanie uruchomione w nowszym środowisku wykonawczym, mogącym wprowadzić dodatkowe i nieobsługiwane przez autorski kod metody w klasy bazowe.

Nadpisana metoda `visit()` przechwytuje wewnętrzną nazwę analizowanej klasy i dokonuje jej translacji na format binarny poprzez zamianę ukośników na kropki, a następnie zapisując rezultat w polu `currentClassName`. Metoda `visitSource()` pozwala na przechwycenie nazwy oryginalnego pliku źródłowego, co jest niezbędne do późniejszego raportowania wykrytych naruszeń struktury API.

Najważniejszym elementem tej warstwy jest nadpisanie metody `visitMethod()`. w momencie gdy silnik ASM identyfikuje deklarację nowej metody w pliku o rozszerzeniu `.class`, wtyczka opakuje domyślny mechanizm w obiekt klasy `DeprecationDetectorMethodVisitor`, przekazując do niego zebrany kontekst klasy i pliku źródłowego. Wymusza to na parserze wejście w obszar instrukcji operacyjnych danej funkcji i uruchamia trzecią warstwę logiki, zaprezentowaną na Listingu 5.11.

Listing 5.11: Przechwytywanie instrukcji wywołań oraz śledzenie linii kodu w `MethodVisitor`. Źródło: opracowanie własne.

```

1 private static class DeprecationDetectorMethodVisitor extends MethodVisitor {
2     private final String callerClass;
3     private final String callerMethod;
4     private final String sourceFile;
5     private final List<DeprecatedUsage> usages;
6     private int currentLineNumber = -1;
7
8     public DeprecationDetectorMethodVisitor(MethodVisitor mv, String callerClass,
9         String callerMethod, String sourceFile, List<DeprecatedUsage> usages) {
10         super(Opcodes.ASM9, mv);
11         this.callerClass = callerClass;
12         this.callerMethod = callerMethod;
13         this.sourceFile = sourceFile;
14         this.usages = usages;
15     }
16
17     @Override
18     public void visitLineNumber(int line, Label start) {
19         this.currentLineNumber = line;
20     }
21 }

```

```

19         super.visitLineNumber(line, start);
20     }
21
22     @Override
23     public void visitMethodInsn(int opcode, String owner, String name, String
        descriptor, boolean isInterface) {
24         if (isMethodDeprecated(owner, name, descriptor)) {
25             String deprecatedClass = owner.replace('/', '.');
26             DeprecatedUsage usage = new DeprecatedUsage(callerClass, callerMethod,
                deprecatedClass, name, descriptor, sourceFile, currentLineNumber
27         );
28             usages.add(usage);
29         }
30         super.visitMethodInsn(opcode, owner, name, descriptor, isInterface);
31     }
32 }

```

Trzecia warstwa realizuje proces detekcji na najniższym poziomie abstrakcji, operując bezpośrednio na instrukcjach sterujących maszyną wirtualną. W konstrukcji tej warstwy kluczową rolę odgrywa przekazanie do instrukcji `super` referencji do nadrzędnego wizytatora, która znajduje się w zmiennej `mv`. Operacja ta aktywuje proces automatycznej delegacji zdarzeń, stanowiący implementację strukturalnego wzorca projektowego Dekorator. Dzięki temu zabiegowi klasa `DeprecationDetectorMethodVisitor` może działać w charakterze bezinwazyjnego filtra. System przechwytuje i poddaje analizie wyłącznie wybrane instrukcje, natomiast wszystkie pozostałe, nieobsługiwane bezpośrednio operacje kodu bajtowego są transparentnie przekazywane do dalszych komponentów, co zapobiega przerwaniu całego łańcucha przetwarzania klasy.

Właściwa ekstrakcja wywołań odbywa się poprzez nadpisanie metody `visitMethodInsn()`. Funkcja ta stanowi centralny punkt detekcji, ponieważ jest automatycznie wyzwalana przez silnik ASM przy każdej napotkanej w kodzie bajtowym instrukcji wywołania metody, niezależnie od tego, czy dotyczy ona standardowej metody obiektowej, statycznej, prywatnej, czy zdefiniowanej w ramach interfejsu. Przekazywany przez parser parametr `opcode` identyfikuje niskopoziomowy rodzaj operacji maszynowej, jednak z punktu widzenia logiki biznesowej systemu wszystkie te zdarzenia są traktowane jednolicie jako potencjalne miejsce wystąpienia przestarzałego API.

Przekazywane przez silnik ASM parametry odzwierciedlają wewnętrzny format struktur JVM. Zmienna `owner` przechowuje nazwę pakietu i klasy docelowej, natomiast `descriptor` reprezentuje skompilowaną i ustrukturyzowaną sygnaturę typów parametrów oraz typu zwracanego. Przykładowo, zapis tekstowy o postaci `(Ljava/lang/String;)V` jednoznacznie identyfikuje metodę przyjmującą jako argument obiekt klasy `String`, która nie zwraca żadnej wartości (`void`).

Precyzyjne odtworzenie lokalizacji analizowanego miejsca w strukturze projektu jest możliwe dzięki mapowaniu linii kodu źródłowego. Kompilator języka Java umieszcza w strukturze pliku binarnego dedykowaną tablicę mapowania o formalnej nazwie `LineNumberTable`, która wiąże instrukcje bajtowe z fizycznymi liniami tekstu napisanego przez programistę. Przechwytując te dane za pomocą metody `visitLineNumber()` wtyczka ma możliwość ciągłej aktualizacji stanu zmiennej `currentLineNumber`. Dzięki temu w momencie identyfikacji przestarzałego wywołania, rejestrowany obiekt klasy `DeprecatedUsage` zostaje zasilony kompletnym zestawem informacji o kontekście wykonania.

Kluczowym aspektem tego modułu jest ścisłe zachowanie zasady rozdzielania odpowiedzialności. Biblioteka ASM pełni w systemie wyłącznie rolę niskopoziomowego parsera strumienia binarnego i sama w sobie nie posiada wbudowanej wiedzy o statusie semantycznym poszczególnych elementów oprogramowania. Silnik ASM w trakcie przekazywania kolejnych instrukcji kodu bajtowego informuje jedynie o fakcie wykrycia relacji wywołania. Decyzja o zakwalifikowaniu danego zdarzenia jako użycia przestarzałego interfejsu została oddelegowana do wewnętrznej metody walidacyjnej `isMethodDeprecated()`, której implementację przedstawia Listing 5.12.

Listing 5.12: Hybrydowa weryfikacja statusu deprecjacji w klasie `BytecodeDeprecationScanner`. Źródło: opracowanie własne.

```
1 private boolean isMethodDeprecated(String owner, String methodName, String descriptor
2 ) {
3     try {
4         String className = owner.replace('/', '.');
5         Class<?> clazz;
6         try {
7             clazz = Class.forName(className, false,
8                 ClassLoader.getSystemClassLoader());
9         } catch (ClassNotFoundException e) {
10             clazz = Class.forName(className, false,
11                 Thread.currentThread().getContextClassLoader());
12         }
13         Class<?>[] paramTypes = parseDescriptorToParamTypes(descriptor);
14         try {
15             Method method = clazz.getDeclaredMethod(methodName, paramTypes);
16             return method.isAnnotationPresent(Deprecated.class);
17         } catch (NoSuchMethodException e) {
18             try {
19                 Method method = clazz.getMethod(methodName, paramTypes);
20                 return method.isAnnotationPresent(Deprecated.class);
21             } catch (NoSuchMethodException ex) {
22                 return checkDeprecationInBytecode(owner, methodName, descriptor);
23             }
24         } catch (ClassNotFoundException | NoClassDefFoundError | SecurityException e) {
25             return checkDeprecationInBytecode(owner, methodName, descriptor);
26         }
27     }
```

Zaimplementowany algorytm realizuje podejście hybrydowe, w którym mechanizm refleksji widoczny w liniach od 4 do 22 pełni rolę wyłącznie wysokopoziomowej optymalizacji, natomiast bezpośrednia analiza kodu bajtowego przy użyciu biblioteki ASM stanowi rdzeń systemu detekcji. Metoda ta przetwarza surowe parametry identyfikacyjne dostarczane bezpośrednio z wizytatora instrukcji maszynowych i decyduje o wyborze najbardziej efektywnej ścieżki weryfikacji.

W pierwszej fazie system podejmuje próbę wykonania szybkiej weryfikacji w pamięci operacyjnej runtime przy użyciu klasy `Class.forName()`. Jest to celowy zabieg optymalizacyjny, który pozwala na natychmiastowe sprawdzenie klas powszechnie dostępnych w środowisku uruchomieniowym (np. podstawowych struktur biblioteki standardowej JDK) bez konieczności kosztownego otwierania strumieni dyskowych. Przekazanie wartości logicznej `false` jako drugiego argumentu tej funkcji blokuje wykonanie statycznych bloków inicjalizacyjnych, zabezpieczając system przed niekontrolowanym uruchomieniem obcego kodu. Ponieważ formaty deskryptorów JVM różnią się od typów wysokopoziomowych, procedura `parseDescriptorToParamTypes()` dokonuje translacji binarnego zapisu sygnatury na tablicę klas obiektowych.

Właściwa faza analizy uruchamia się w momencie, gdy badana struktura nie jest załadowana w pamięci JVM, co stanowi naturalny stan dla klas skanowanego projektu użytkownika. Po przechwyceniu wyjątku o braku klasy, system rezygnuje z warstwy refleksyjnej i przekazuje sterowanie do metody `checkDeprecationInBytecode()`. W tej fazie program uruchamia pełny potok biblioteki ASM, pobierając surowy strumień pliku o rozszerzeniu `.class` bezpośrednio z katalogów wyjściowych procesu budowania i przeszukując jego metadane przy pomocy wizytatora `DeprecationCheckVisitor`. Takie podejście gwarantuje, że system nie jest ograniczony wyłącznie do zasobów załadowanych w pamięci podręcznej środowiska, a operacja bezpośredniego czytania bajtów za pomocą ASM stanowi główny strumień analizy skanera. Pełną implementację tego mechanizmu dekodowania struktur binarnych zaprezentowano na Listingu 5.13.

Listing 5.13: Implementacja metody `checkDeprecationInBytecode` oraz wizytatora `DeprecationCheckVisitor`. Źródło: opracowanie własne.

```

1 private boolean checkDeprecationInBytecode(String owner, String methodName, String
  descriptor) {
2     try {
3         String resourceName = owner + ".class";
4         InputStream is = Thread.currentThread().getContextClassLoader().
5             getResourceAsStream(resourceName);
6
7         if (is == null) {
8             return false;
9         }
10        try (is) {
11            ClassReader reader = new ClassReader(is);
12            DeprecationCheckVisitor visitor = new DeprecationCheckVisitor(methodName,
13                descriptor);
14            reader.accept(visitor, ClassReader.SKIP_CODE | ClassReader.SKIP_FRAMES);
15            return visitor.isDeprecated();
16        } catch (IOException e) {
17            return false;
18        }
19    }
20
21    private static class DeprecationCheckVisitor extends ClassVisitor {
22        private final String targetMethodName;
23        private final String targetDescriptor;
24        private boolean deprecated = false;
25
26        public DeprecationCheckVisitor(String methodName, String descriptor) {
27            super(Opcodes.ASM9);
28            this.targetMethodName = methodName;
29            this.targetDescriptor = descriptor;
30        }
31
32        @Override
33        public MethodVisitor visitMethod(int access, String name, String descriptor,
34            String signature, String[] exceptions) {
35            if (name.equals(targetMethodName) && descriptor.equals(targetDescriptor)) {
36                return new MethodVisitor(Opcodes.ASM9) {
37                    @Override
38                    public AnnotationVisitor visitAnnotation(String desc, boolean visible)
39                        {
40                            if (desc.equals("Ljava/lang/Deprecated;")) {
41                                deprecated = true;
42                            }
43                            return super.visitAnnotation(desc, visible);
44                        }
45                };
46            }
47            return null;
48
49            public boolean isDeprecated() {
50                return deprecated;
51            }
52    }

```

Analiza przedstawionego algorytmu pozwala na zidentyfikowanie technik optymalizacji strumieniowej oferowanych przez architekturę ASM. Metoda `checkDeprecationInBytecode()` dynamicznie mapuje wewnętrzną nazwę klasy na ścieżkę zasobu sieciowego lub dyskowego poprzez dodanie właściwego rozszerzenia binarnego. Pobranie surowego strumienia danych realizowane jest za pomocą metody

`getResourceAsStream()`, co pozwala na odczytanie zawartości pliku bez pośrednictwa mechanizmu ładowania klas i bez narzutu na pamięć operacyjną maszyny wirtualnej.

Kluczowym aspektem tej procedury jest konfiguracja metody `accept()` w klasie `ClassReader`. Po przez zastosowanie koniunkcji bitowej flag `ClassReader.SKIP_CODE` oraz `ClassReader.SKIP_FRAMES`, parser całkowicie pomija przetwarzanie instrukcji logicznych wewnątrz metod, struktur sterujących, pętli oraz tablic lokalnych ramek stosu. Silnik ogranicza swoje działanie wyłącznie do odczytu sygnatur i struktur nagłówkowych.

Detekcja adnotacji realizowana jest w klasie `DeprecationCheckVisitor`, która selektywnie analizuje strukturę pliku. Nadpisana metoda `visitMethod()` dokonuje natychmiastowej ewaluacji nazwy oraz deskryptora napotkanego komponentu. Jeśli parametry nie są tożsame z poszukiwaną sygnaturą, funkcja zwraca wartość pustą, co natychmiastowo przerywa dalsze dekodowanie danego bloku kodu. W sytuacji pełnej zgodności, system inicjalizuje anonimowy obiekt klasy `MethodVisitor` i nadpisuje metodę `visitAnnotation()`. Przechwycenie deskryptora adnotacji o strukturze `Ljava/lang/Deprecated;` skutkuje przedstawieniem wewnętrznej flagi stanu, która po zakończeniu cyklu parsującego jest zwracana jako ostateczny wynik weryfikacji. Tak zaprojektowana architektura gwarantuje maksymalną odporność i szybkość działania systemu, stanowiąc komplementarną i całkowicie niezależną od środowiska uruchomieniowego ścieżkę weryfikacji metadanych binarnego kodu źródłowego.

5.6. Integracja z modelem AI i inżynieria zapytań

Kluczowym elementem warstwy analitycznej wtyczki jest integracja z dużym modelem językowym (ang. *Large Language Model*), który odpowiada za generowanie bezpiecznych i kontekstowych propozycji refaktoryzacji przestarzałego kodu. W obecnej architekturze system integruje się z modelem Google Gemini 2.5 Flash, udostępnianym za pośrednictwem oficjalnego interfejsu REST API. Całość logiki komunikacyjnej oraz proces konstruowania zapytań hermetyzuje klasa `AIRefactoringService`.

W celu uzyskania jak najdokładniejszej odpowiedzi system wykorzystuje technikę dynamicznego wstrzykiwania kontekstu, budując precyzyjne zapytanie (ang. *prompt*) w oparciu o metadane zgromadzone przez warstwy skanujące (opisane w podrozdziałach 5.3 i 5.4). Listing 5.14 prezentuje fragment odpowiedzialny za inżynierię zapytania.

Listing 5.14: Konstrukcja zapytania do modelu AI z wykorzystaniem zebranego kontekstu. Źródło: opracowanie własne.

```
1 private String buildPrompt(MethodUsageInfo usage, String codeContext) {
2     StringBuilder prompt = new StringBuilder();
3     prompt.append("You are a Java code refactoring expert. A deprecated method is
4         being used and needs to be replaced.\n\n");
5     prompt.append("Deprecated Method Information:\n");
6     prompt.append("- Class: ").append(usage.deprecatedMethod().className()).
7         append("\n");
8     prompt.append("- Method: ").append(usage.deprecatedMethod().methodName()).
9         append("\n");
10    prompt.append("- Signature: ").append(usage.deprecatedMethod().
11        methodSignature()).append("\n");
12
13    if (!usage.deprecatedMethod().deprecationReason().isEmpty()) {
14        prompt.append("- Deprecation Reason: ").append(usage.deprecatedMethod().
15            deprecationReason()).append("\n");
16    }
17
18    if (usage.deprecatedMethod().hasSuggestedReplacements()) {
19        prompt.append("- Suggested Replacements: ").append(String.join(", ",
20            usage.deprecatedMethod().suggestedReplacements())).append("\n");
21    }
22 }
```

```

17     prompt.append("\nCode Context:\n");
18     prompt.append("```\n");
19     prompt.append(codeContext);
20     prompt.append("\n```\n");
21
22     prompt.append("Please provide:\n");
23     prompt.append("1. The EXACT refactored code that replaces the deprecated
24         method call (just the fixed code, no explanations)\n");
25     prompt.append("2. After '---EXPLANATION---', provide a brief explanation of
26         the changes\n");
27     prompt.append("Format your response as:\n");
28     prompt.append("REFACTORED CODE:\n");
29     prompt.append("```\n");
30     prompt.append("[your refactored code here]\n");
31     prompt.append("```\n");
32     prompt.append("---EXPLANATION---\n");
33     prompt.append("[your explanation here]\n");
34
35     return prompt.toString();
36 }

```

W zaprezentowanym kodzie wtyczka precyzyjnie definiuje rolę modelu (ang. *Role Prompting*) oraz dostarcza mu nie tylko sam kod do poprawy, ale przede wszystkim powód deprecjacji oraz listę sugerowanych zamienników wyekstrahowanych z dokumentacji Javadoc. Szczegółowe zestawienie wszystkich komponentów wstrzykiwanych do zapytania oraz ich bezpośrednie źródła w strukturach danych wtyczki przedstawiono w Tabeli 5.1.

Tabela 5.1: Składniki kontekstu wstrzykiwane do zapytania dla modelu AI. Źródło: opracowanie własne.

Element promptu	Źródło danych w kodzie	Znaczenie
Nazwa klasy	<code>usage.deprecatedMethod().className()</code>	Określa pełny kontekst typu i pochodzenie analizowanej metody.
Nazwa metody	<code>usage.deprecatedMethod().methodName()</code>	Wskazuje konkretny identyfikator usuwanego wywołania.
Pełna sygnatura	<code>usage.deprecatedMethod().methodSignature()</code>	Precyzuje typy argumentów, co zapobiega halucynacjom w przypadku metod przeciążonych.
Powód deprecacji	<code>usage.deprecatedMethod().deprecationReason()</code>	Dostarcza modelowi tekstowy opis intencji autora biblioteki wyciągnięty z tagu <code>@deprecated</code> .
Sugerowane zamienniki	<code>usage.deprecatedMethod().suggestedReplacements()</code>	Lista znaczników <code>{@link}</code> pobrana z drzewa AST Javadoc, stanowiąca preferowaną ścieżkę naprawy.
Kontekst kodu	<code>Document.getText(TextRange)</code>	Wycinek edytora (± 5 linii), dający modelowi niezbędny kontekst zmiennych i logiki biznesowej.

Co niezwykle istotne z punktu widzenia platformy IntelliJ, fragment kodu źródłowego wstrzykiwany jako otoczenie wywołania i reprezentowany przez zmienną `codeContext` pobierany jest bezpośrednio z wirtualnego edytora przy pomocy metody `Document.getText(TextRange)`. Gwarantuje to, że model LLM analizuje zawsze aktualny stan kodu, nawet jeśli użytkownik wprowadził modyfikacje i nie zapisał jeszcze pliku na dysku. W celu umożliwienia automatycznego parsowania wyników prompt narzuca modelowi rygorystyczny szablon wyjściowy, wymagający ścisłego podziału na sekcję kodu o nazwie `REFACTORED CODE` oraz opis słowny zdefiniowany jako `EXPLANATION`.

Aby zilustrować ostateczny rezultat działania metody `buildPrompt` oraz mechanizmu wstrzykiwania kontekstu, na Listingu 5.15 zaprezentowano rzeczywiste zapytanie wysyłane do API modelu językowego. Przykład ten dotyczy częstego scenariusza refaktoryzacji, w którym metoda przyjmująca

parametr tekstowy `String` została zastąpiona nowszą, wykorzystującą bezpieczne typy wyliczeniowe `Enum`. Dostarczenie modelowi sztucznej inteligencji precyzyjnych metadanych, w szczególności sugerowanego zamiennika oraz samego wywołania z konkretnym argumentem, pozwala mu na wykonanie inteligentnego mapowania. LLM nie musi zgadywać intencji programisty, lecz potrafi na podstawie kontekstu przekonwertować dotychczasową wartość tekstową (np. "ACTIVE") na odpowiednią stałą z nowego typu wyliczeniowego.

Listing 5.15: Przykład wygenerowanego zapytania (promptu) wysyłanego do LLM. Źródło: opracowanie własne.

```
1 You are a Java code refactoring expert. A deprecated method is being used and needs
  to be replaced.
2
3 - Class: com.example.services.AccountService
4 - Method: updateStatus
5 - Signature: void updateStatus(java.lang.String)
6 - Deprecation Reason: String-based statuses are error-prone. Use enum instead.
7 - Suggested Replacements: updateStatus(com.example.enums.AccountStatus)
8
9 Code Context:
10 ```java
11 public void processUserAccount(User user) {
12     logger.info("Processing user: " + user.getId());
13     accountService.updateStatus("ACTIVE");
14     notifier.sendEmail(user);
15 }
16 ```
17
18 Please output your response using EXACTLY the following format:
19 REFACTORED CODE:
20 ```java
21 // your code here
22 ```
23 EXPLANATION:
24 // your explanation here
```

Dzięki tak sformułowanej instrukcji tzw. *Zero-Shot Prompting* z wymuszeniem szablonu, odpowiedź modelu jest wysoce przewidywalna. System ignoruje ewentualny nadmiarowy tekst generowany przez LLM przed blokiem kodu, a parser wtyczki może bezpiecznie wyekstrahować nową logikę. Przykładową, poprawną odpowiedź modelu na powyższe zapytanie przedstawiono na Listingu 5.16.

Listing 5.16: Przykładowa odpowiedź wygenerowana przez model sztucznej inteligencji. Źródło: opracowanie własne.

```
1 REFACTORED CODE:
2 ```java
3 public void processUserAccount(User user) {
4     logger.info("Processing user: " + user.getId());
5     accountService.updateStatus(com.example.enums.AccountStatus.ACTIVE);
6     notifier.sendEmail(user);
7 }
8 ```
9 EXPLANATION:
10 The deprecated method updateStatus(String) was replaced with the suggested
    updateStatus(AccountStatus) to ensure type safety. The string literal "ACTIVE"
    was successfully mapped to the AccountStatus.ACTIVE enum value.
```

Po asynchronicznym odebraniu powyższej odpowiedzi, logika wtyczki rozdziela tekst na podstawie zdefiniowanych słów kluczowych `REFACTORED CODE` oraz `EXPLANATION`. Wyodrębniony blok kodu jest następnie parsowany, a sekcja z wyjaśnieniem trafia do interfejsu graficznego, pomagając programiście

w podjęciu decyzji o akceptacji zmian.

5.7. Transformacja kodu i modyfikacja drzewa PSI

Ostatnim etapem procesu analizy jest fizyczne naniesienie wygenerowanych lub zdeterminowanych poprawek na kod źródłowy projektu. W architekturze platformy IntelliJ IDEA modyfikacja plików nie może odbywać się w sposób bezpośredni za pomocą standardowych operacji wejścia/wyjścia. Zamiast tego wtyczka musi korzystać ze mechanizmów zarządzania modelem dokumentu oraz strukturalnym drzewem PSI.

Kluczowym wymogiem bezpieczeństwa jest realizacja wszelkich modyfikacji wewnątrz tak zwanej akcji zapisu (ang. *Write Action*). Klasy odpowiedzialne za transformację wykorzystują w tym celu fasadę `WriteCommandAction`, co gwarantuje spełnienie trzech fundamentalnych warunków stabilności środowiska programistycznego.

- **Blokada równoległych operacji odczytu** – uniemożliwia innym wątkom tła jednoczesny dostęp do modyfikowanych struktur danych, co zapobiega powstawaniu błędów typu *race condition* w pamięciowym modelu środowiska programistycznego.
- **Zapewnienie pełnej atomowości transakcji** – gwarantuje, że wszystkie zaplanowane zmiany w strukturze kodu aplikowane są jako jedna niepodzielna operacja, co chroni projekt przed pozostawieniem w stanie niespójnym w przypadku wystąpienia nieoczekiwanej awarii.
- **Automatyczna integracja z systemem historii** – rejestruje wprowadzone modyfikacje w centralnym menedżerze operacji środowiska, umożliwiając użytkownikowi bezproblemowe cofanie oraz ponawianie zmian (ang. *Undo/Redo*) bezpośrednio z poziomu edytora graficznego.

System wtyczki implementuje dwie odrębne ścieżki modyfikacji kodu w zależności od pochodzenia danych refaktoryzacyjnych. Pierwsza z nich, zaimplementowana w klasie `RefactoringService`, obsługuje ścieżkę zamienników pozyskanych bezpośrednio z metadanych lub znaczników dokumentacyjnych. Główne operacje transformacji węzłów prezentuje Listing 5.17.

Listing 5.17: Orkiestracja transformacji strukturalnych w klasie `RefactoringService`. Źródło: opracowanie własne.

```
1 public RefactoringResult applyRefactoring(MethodUsageInfo usage, RefactoringAction
   action) {
2     try {
3         PsiFile psiFile = findPsiFile(usage.fullFilePath());
4         if (psiFile == null) {
5             return new RefactoringResult(false, "Could not find PSI file for: " +
               usage.fullFilePath());
6         }
7
8         PsiElement element = ApplicationManager.getApplication().runReadAction(
9             (com.intellij.openapi.util.Computable<PsiElement>) () ->
               findMethodCallElement(psiFile, usage)
10        );
11
12        if (element == null) {
13            return new RefactoringResult(false, "Could not find method call element
               at line " + usage.lineNumber());
14        }
15
16        return switch (action.type()) {
17            case REPLACE_METHOD_CALL -> replaceMethodCall(element, action, usage);
18            case ADD_SUPPRESSION -> addSuppressionAnnotation(element, action);
```

```

19         case ADD_TODO_COMMENT -> addTodoComment(element, action);
20     };
21     } catch (Exception e) {
22         return new RefactoringResult(false, "Error during refactoring: " + e.
                getMessage());
23     }
24 }

```

Prezentowana metoda `applyRefactoring()` pełni funkcję interfejsu wykonawczego dla systemu automatycznych poprawek udostępnianych użytkownikowi w oknie inspekcji edytora. Głównym celem użytkowym tego komponentu jest dostarczenie programiście wielowariantowego mechanizmu decyzyjnego w obszarze zarządzania długiem technologicznym.

W zależności od kontekstu projektowego system udostępnia za pośrednictwem instrukcji wyboru trzy funkcjonalności aplikacyjne:

- **Automatyczna migracja** powiązana z wariantem `REPLACE_METHOD_CALL` pozwala na natychmiastowe zastąpienie przestarzałego wywołania nowym, rekomendowanym przez twórców biblioteki odpowiednikiem. Jest to podstawowy scenariusz naprawczy stosowany w sytuacji, gdy dokumentacja techniczna wprost definiuje bezpośredni i bezpieczny zamiennik API.
- **Odłożenie refaktoryzacji** wyzwalane opcją `ADD_SUPPRESSION` służy do automatycznego wstrzyknięcia adnotacji wyciszającej ostrzeżenia kompilatora bezpośrednio nad sygnaturą metody nadrzędnej. Funkcjonalność ta znajduje zastosowanie w krytycznych fazach wdrożeniowych, gdy modyfikacja logiki biznesowej niesie ze sobą zbyt duże ryzyko regresji, a priorytetem jest oczyszczenie raportów kompilacji.
- **Ewidencja długu technologicznego** realizowana przez instrukcję `ADD_TODO_COMMENT` wprowadza bezpośrednio przed analizowanym wywołaniem ustrukturyzowany komentarz techniczny. Opcja ta jest wykorzystywana w pracy zespołowej do znakowania miejsc wymagających późniejszej interwencji programistycznej bez jednoczesnego wpływu na stabilność i działanie aktualnego kodu binarnego.

Druga ścieżka modyfikacji kodu, realizowana przez klasę `AIRefactoringService`, odpowiada za obsługę poprawek generowanych dynamicznie przez model sztucznej inteligencji. Z uwagi na fakt, że generowanie odpowiedzi przez zewnętrzną usługę sieciową wiąże się z opóźnieniami, architektura tego komponentu została podzielona na dwa powiązane etapy, którymi są asynchroniczne pozyskiwanie sugestii oraz niskopoziomowa transformacja dokumentu tekstowego. Pierwszą fazę, odpowiedzialną za orkiestrację procesu asynchronicznego oraz pobieranie kontekstu kodu, przedstawia Listing 5.18.

Listing 5.18: Asynchroniczne pobieranie kontekstu i generowanie sugestii sztucznej inteligencji w klasie `AIRefactoringService`. Źródło: opracowanie własne.

```

1 public void suggestAndApplyRefactoring(MethodUsageInfo usage,
2     RefactoringSuggestionCallback callback) {
3     ProgressManager.getInstance().run(new Task.Backgroundable(project, "Generating AI
4         Refactoring Suggestion", true) {
5         private String suggestion;
6         private String explanation;
7
8         @Override
9         public void run(@NotNull ProgressIndicator indicator) {
10             indicator.setText("Reading code context...");
11             String codeContext = ApplicationManager.getApplication().runReadAction(
12                 (com.intellij.openapi.util.Computable<String>) () -> getCodeContext(
13                     usage)
14             );
15
16             if (codeContext == null) {

```

```

14         ApplicationManager.getApplication().invokeLater(() -> callback.
15             onError("Could not read code context"));
16     return;
17 }
18
19 indicator.setText("Generating AI suggestion...");
20 try {
21     String[] result = generateAISuggestion(usage, codeContext, indicator)
22     ;
23     suggestion = result[0];
24     explanation = result[1];
25
26     if (suggestion == null || suggestion.trim().isEmpty()) {
27         ApplicationManager.getApplication().invokeLater(() ->
28             callback.onError("AI did not generate a valid suggestion")
29         );
30         return;
31     }
32
33     ApplicationManager.getApplication().invokeLater(() -> callback.
34         onSuccess(suggestion, explanation, codeContext));
35 } catch (Exception e) {
36     ApplicationManager.getApplication().invokeLater(() -> callback.
37         onError("Error generating AI suggestion: " + e.getMessage()));
38 }
39 }
40 }
41 }
42 }
43 }

```

Metoda `suggestAndApplyRefactoring()` stanowi fundament dla inteligentnego systemu wspomagania rozwoju oprogramowania opartego na modelach generatywnych. Głównym zadaniem użytkowym tego komponentu jest automatyzacja skomplikowanych i niedeterministycznych procesów analizy kodu źródłowego w sytuacjach, gdy proste podstawienie nazwy metody okazuje się niewystarczające.

W obszarze interakcji z programistą oraz zarządzania kodem użytkownika omawiany serwis realizuje trzy kluczowe funkcjonalności aplikacji.

- **Generowanie kontekstowych scenariuszy naprawczych** – umożliwia obsługę wielolinowych transformacji i adaptację struktur kodu do nowoczesnych wzorców projektowych na podstawie analizy semantycznej. Jest to kluczowe zastosowanie w przypadku złożonych bibliotek, gdzie usunięcie danej metody wymaga jednoczesnej zmiany logiki inicjalizacji obiektów powiązanych.
- **Dostarczanie wyjaśnień inżynierskich** – oprócz wygenerowania surowego bloku naprawczego system pozyskuje od modelu sztucznej inteligencji ustrukturyzowane uzasadnienie wprowadzonych zmian. Pomaga to programiście w szybkim zrozumieniu intencji nowego API oraz pełni funkcję edukacyjną w procesie ciągłej eliminacji długu technologicznego.
- **Interaktywny mechanizm weryfikacji poprawek** – dzięki asynchronicznemu przekazywaniu wyników do interfejsu użytkownika system umożliwia wyświetlenie bezpiecznego podglądu zmian przed ich fizycznym zatwierdzeniem w pliku, co minimalizuje ryzyko wprowadzenia niepożądanych efektów ubocznych.

Po pomyślnym wygenerowaniu rekomendacji oraz uzyskaniu akceptacji użytkownika system przechodzi do drugiej fazy, którą jest fizyczne zmodyfikowanie zawartości pliku. Z uwagi na fakt, że wynik działania dużych modeli językowych często obejmuje modyfikację wielu linii kodu i przekracza granice pojedynczego węzła AST, zastosowano w niej odmienne podejście. Zamiast operować na strukturach wygenerowanych przez fabrykę PSI, system wykonuje niskopoziomowe operacje bezpośrednio na obiekcie klasy `Document`. Główną procedurę odpowiedzialną za aplikowanie poprawek tekstowych oraz transakcyjną synchronizację prezentuje Listing 5.19.

Listing 5.19: Modyfikacja dokumentu tekstowego z wymuszeniem synchronizacji PSI w klasie `AIRefactoringService`. Źródło: opracowanie własne.

```
1 public void applyRefactoring(MethodUsageInfo usage, String newCode) {
2     PsiFile psiFile = findPsiFile(usage.fullFilePath());
3     if (psiFile == null) {
4         Messages.showErrorDialog(project, "Could not find file: " + usage.
5             fullFilePath(), "Refactoring Error");
6         return;
7     }
8     WriteCommandAction.runWriteCommandAction(
9         project,
10        "AI Refactoring",
11        "RefactoringPlugin",
12        () -> {
13            try {
14                PsiDocumentManager documentManager = PsiDocumentManager.getInstance(
15                    project);
16                com.intellij.openapi.editor.Document document = documentManager.
17                    getDocument(psiFile);
18
19                if (document == null || usage.lineNumber() <= 0) {
20                    return;
21                }
22
23                int startLine = Math.max(0, usage.lineNumber() - 6);
24                int endLine = Math.min(document.getLineCount() - 1, usage.lineNumber
25                    () + 4);
26
27                int startOffset = document.getLineStartOffset(startLine);
28                int endOffset = document.getLineEndOffset(endLine);
29
30                document.replaceString(startOffset, endOffset, newCode);
31                documentManager.commitDocument(document);
32
33                CodeStyleManager.getInstance(project).reformatText(psiFile,
34                    startOffset, startOffset + newCode.length());
35
36            } catch (Exception e) {
37                Messages.showErrorDialog(project, "Failed to apply refactoring: " + e
38                    .getMessage(), "Refactoring Error");
39            }
40        }
41    );
42 }
```

Prezentowana metoda `applyRefactoring()` stanowi dopełnienie potoku sztucznej inteligencji, odpowiadając za ostateczne zatwierdzenie i utrwalenie zmian w strukturze projektu. Wdrożenie bezpośredniej operacji na tekście wymusiło zastosowanie specyficznych mechanizmów platformy IntelliJ, których cechy operacyjne opisują następujące punkty:

- **Manipulacja buforem** – system całkowicie rezygnuje z cząstkowej manipulacji pojedynczymi węzłami drzewa składniowego i wykonuje operację bezpośredniego zastąpienia całego bloku znaków za pomocą metody `replaceString()` na obiekcie klasy `Document`, co umożliwia stabilne wstrzykiwanie złożonych i wieloliniowych struktur kodu.
- **Wymuszenie synchronizacji struktur** – wywołanie metody `commitDocument()` na menedżerze typu `PsiDocumentManager` stanowi krytyczny krok synchronizacyjny, który zmusza środowisko do ponownego sparsowania zaktualizowanego fragmentu tekstu i odbudowania lokalnego poddrzewa PSI, co zapobiega rozspójnieniu widoku edytora z wewnętrznym modelem programu.

- **Dynamiczne obliczanie zakresów przesunięć** – za pomocą metod `getLineStartOffset()` oraz `getLineEndOffset()` program wyznacza granice fizycznego bloku tekstu przeznaczonego do zastąpienia na podstawie numeru linii zarejestrowanego w obiekcie klasy `MethodUsageInfo`.

Ostatnią fazą dla obu niezależnych ścieżek transformacji jest końcowe formatowanie wstawionego kodu. Wtyczka deleguje to zadanie do systemowej klasy `CodeStyleManager`, która aplikuje reguły zgodne z indywidualnymi preferencjami użytkownika środowiska programistycznego. Wybór właściwego podejścia zależy bezpośrednio od sposobu reprezentacji zmodyfikowanego fragmentu kodu. Metoda `reformat()` jest stosowana w sytuacjach, gdy system operuje na konkretnym i kompletnym obiekcie strukturalnym drzewa składniowego, takim jak cała deklaracja klasy czy pojedynczy blok instrukcji. Z kolei metoda `reformatText()` służy do przetwarzania zmian opisanych przedziałami liczbowymi, które określają dokładną pozycję początku i końca modyfikowanego ciągu znaków w strukturze dokumentu.

5.8. Warstwa prezentacji i interfejsu użytkownika

Warstwa prezentacji odpowiada za interakcję z programistą i wyświetlanie wyników analizy. W celu zachowania pełnej spójności wizualnej ze środowiskiem IntelliJ IDEA, interfejs graficzny wtyczki został zbudowany z wykorzystaniem komponentów dostarczanych przez framework graficzny platformy JetBrains. Podstawowe kontenery i układy okien bazują na klasycznej bibliotece Java Swing, zaimplementowanej między innymi poprzez klasę `JPanel`, jednak standardowe elementy interfejsu zastąpiono ich dedykowanymi odpowiednikami z platformy IntelliJ. Główne komponenty wykorzystane przy realizacji interfejsu wtyczki oraz ich bezpośrednie przeznaczenie zestawiono w Tabeli 5.2.

Tabela 5.2: Zestawienie komponentów interfejsu graficznego wykorzystanych we wtyczce. Źródło: opracowanie własne.

Komponent IntelliJ	Odpowiednik Swing	Zastosowanie we wtyczce
<code>JTable</code>	<code>JTable</code>	Wyświetlanie listy wykrytych użyć metod oraz zależności.
<code>JScrollPane</code>	<code>JScrollPane</code>	Przewijanie zawartości tabel i paneli tekstowych.
<code>JLabel</code>	<code>JLabel</code>	Etykiety tekstowe w oknach i panelach informacyjnych.
<code>DialogWrapper</code>	<code>JDialog</code>	Klasa bazowa do tworzenia standardowych okien dialogowych.

W celu zapewnienia płynnej integracji z ekosystemem JetBrains, architektura interfejsu graficznego systemu została zadeklarowana strukturalnie w pliku konfiguracyjnym wtyczki za pomocą dwóch odrębnych punktów integracji.

- **Dedykowane okno narzędziowe** projektowane poprzez implementację klasy `RefactoringPluginToolWindowFactory` rejestruje panel boczny umieszczony po prawej stronie przestrzeni roboczej środowiska deweloperskiego. W tym obszarze znajduje się główna tabela klasy `JTable`, która agreguje wyniki skanowania kodu i pozwala programiście na wygodne zarządzanie całym wykrytym długim technologicznym z poziomu jednego zestawienia.
- **Panel konfiguracyjny w ustawieniach środowiska** realizowany przez komponent `RefactoringPluginConfigurable` osadza dedykowaną sekcję opcji w menu narzędziowym platformy IntelliJ. Moduł ten ściśle współpracuje z usługą aplikacyjną `RefactoringPluginSettings` i dostarcza użytkownikowi bezpieczny interfejs do zmiany ustawień.

Podczas projektowania interfejsu graficznego kluczowym aspektem było zapewnienie, aby długotrwałe operacje sieciowe nie blokowały głównego wątku dystrybucji zdarzeń EDT, co powodowałoby natychmiastowe zawieszenie całego środowiska deweloperskiego. Problem ten rozwiązano poprzez wdrożenie asynchronicznego modelu wymiany danych, bazującego na dwóch powiązanych ze sobą procesach.

- **Wykonywanie ciężkich zadań w tle** – wszelkie czasochłonne operacje, są delegowane do wątków pobocznych za pomocą klasy `ProgressManager` oraz zadania `Task.Backgroundable`, co pozwala użytkownikowi na bieżąco śledzić pasek postępu oraz daje możliwość bezpiecznego anulowania operacji w dowolnym momencie.
- **Aktualizacja komponentów graficznych z wątku głównego** – po zakończeniu procedur obliczeniowych w tle, finalne wyniki są przekazywane z powrotem do elementów interfejsu użytkownika za pomocą metody `invokeLater()` wywoływanej na obiekcie klasy `ApplicationManager`, co gwarantuje pełne bezpieczeństwo wątkowe operacji graficznych.

Jedną z najważniejszych funkcjonalności interfejsu użytkownika podnoszących bezpieczeństwo refaktoryzacji jest okno podglądu zmian, zaimplementowane w klasie `RefactoringPreviewDialog`. Przed fizycznym naniesieniem poprawek programista ma możliwość zweryfikowania proponowanej modyfikacji w ujednoliconym oknie porównawczym. Logikę tworzenia tego strukturalnego panelu przedstawia Listing 5.20.

Listing 5.20: Tworzenie panelu podglądu różnicowego. Źródło: opracowanie własne.

```

1 private JPanel createDiffPanel() {
2     JPanel panel = new JPanel(new BorderLayout());
3
4     DiffContentFactory contentFactory = DiffContentFactory.getInstance();
5     DocumentContent content1 = contentFactory.create(
6         project, originalCode, PlainTextFileType.INSTANCE);
7     DocumentContent content2 = contentFactory.create(
8         project, suggestedCode, PlainTextFileType.INSTANCE);
9
10    SimpleDiffRequest request = new SimpleDiffRequest(
11        "Refactoring Preview",
12        content1, content2,
13        "Original Code", "Suggested Code"
14    );
15
16    DiffRequestPanel diffPanel = DiffManager.getInstance()
17        .createRequestPanel(project, getDisposable(), null);
18    diffPanel.setRequest(request);
19
20    panel.add(diffPanel.getComponent(), BorderLayout.CENTER);
21    return panel;
22 }

```

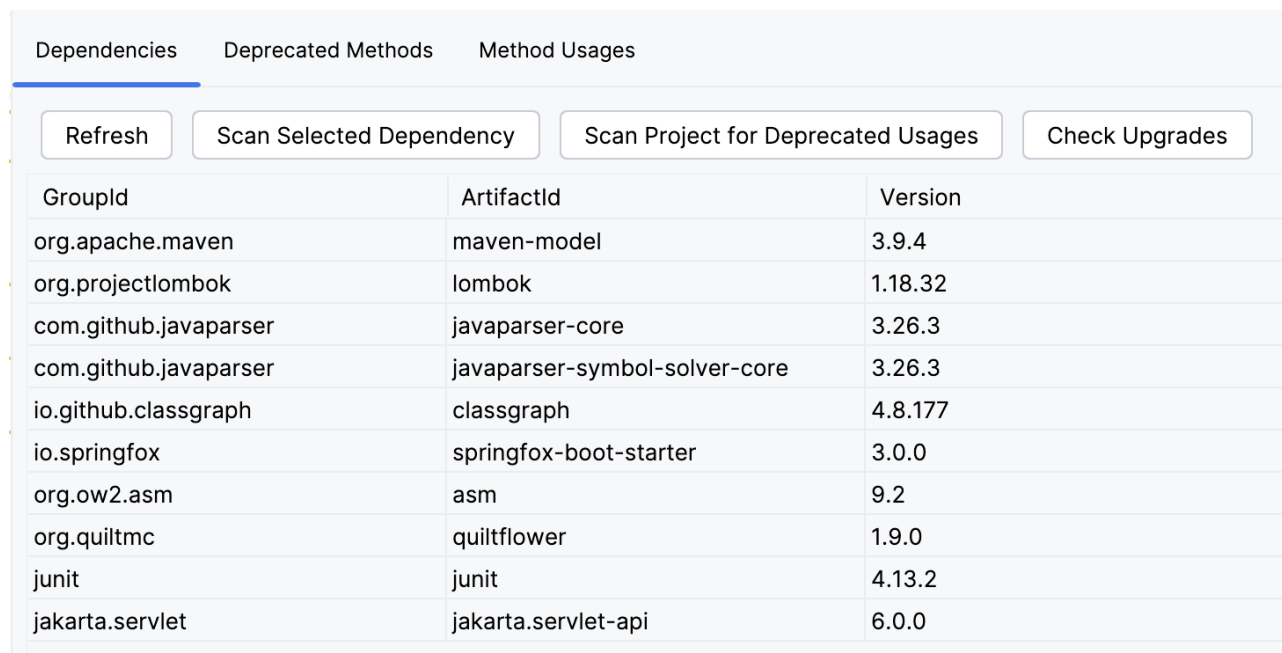
Mechanizm ten wykorzystuje API platformy reprezentowane przez klasę `DiffManager`. Tekst oryginalny oraz propozycja odebrana z modelu językowego są zamieniane na obiekty klasy `DocumentContent`, a następnie wyświetlane obok siebie w widoku porównawczym środowiska IntelliJ, który automatycznie koloruje i oznacza usunięte oraz dodane linie kodu. Powiązanie cyklu życia panelu różnicowego z oknem dialogowym poprzez przekazanie metody `getDisposable()` zapewnia poprawne i automatyczne zwalnianie zasobów pamięciowych przez system. W zależności od decyzji użytkownika podejmowanej w tym oknie za pomocą metody `showAndGet()`, wtyczka albo przerywa zaplanowane działanie, albo uruchamia transakcyjny proces modyfikacji pliku źródłowego opisany szczegółowo w podrozdziale 5.7.

6. Przykładowe wykorzystanie prototypu

W niniejszym rozdziale przedstawiono praktyczne aspekty funkcjonowania opracowanego prototypu wtyczki dedykowanej dla środowiska IntelliJ IDEA. Głównym celem tej części dokumentu jest ukazanie sposobu w jaki aplikacja oraz powiązane z nią moduły realizują procesy detekcji i eliminacji przestarzałych zależności w projekcie programistycznym.

6.1. Wizualizacja głównego interfejsu

Prezentacja działania systemu rozpoczyna się od wyświetlenia głównego widoku wtyczki zintegrowanego z obszarem roboczym środowiska IntelliJ IDEA. Rzeczywisty wygląd okna narzędzia po inicjalizacji przedstawia Rysunek 6.1. Panel ten udostępnia interaktywną tabelę prezentującą zestawienie zależności zidentyfikowanych bezpośrednio w plikach konfiguracyjnych projektu. Lista ta zawiera informacje o grupie, identyfikatorze artefaktu oraz aktualnej wersji każdego pakietu zewnętrznego.



The screenshot shows the main panel of the plugin with three tabs: 'Dependencies' (selected), 'Deprecated Methods', and 'Method Usages'. Below the tabs are four buttons: 'Refresh', 'Scan Selected Dependency', 'Scan Project for Deprecated Usages', and 'Check Upgrades'. A table displays the following data:

GroupId	ArtifactId	Version
org.apache.maven	maven-model	3.9.4
org.projectlombok	lombok	1.18.32
com.github.javaparser	javaparser-core	3.26.3
com.github.javaparser	javaparser-symbol-solver-core	3.26.3
io.github.classgraph	classgraph	4.8.177
io.springfox	springfox-boot-starter	3.0.0
org.ow2.asm	asm	9.2
org.quiltmc	quiltflower	1.9.0
junit	junit	4.13.2
jakarta.servlet	jakarta.servlet-api	6.0.0

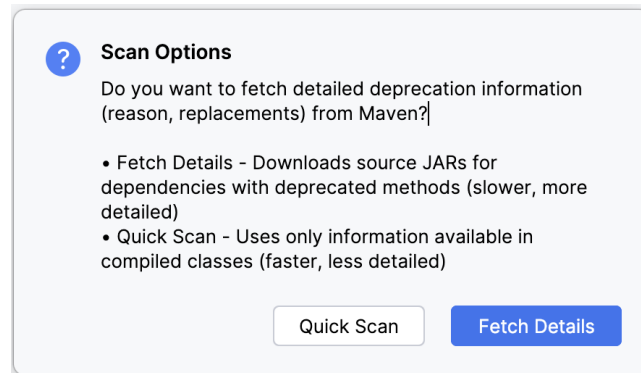
Rysunek 6.1: Główny panel wtyczki z widoczną tabelą zależności i przyciskami funkcyjnymi. Źródło: opracowanie własne.

Główny interfejs wtyczki został zaprojektowany w sposób umożliwiający pełną kontrolę nad procesem analizy. Po zmianie konfiguracji projektu użytkownik ma możliwość odświeżenia widoku tabeli poprzez kliknięcie przycisku **Refresh**. Centralną część interfejsu zajmują zakładki organizujące wyniki analizy, w tym zestawienie przestarzałych metod istniejących w kodzie wybranej zależności (**Deprecated Methods**) lub ich bezpośrednie użycia w kodzie źródłowym (**Method Usages**).

Z poziomu głównego okna użytkownik może zainicjować dwa odrębne tryby skanowania. Wybranie opcji **Scan Selected Dependency** uruchamia analizę zewnętrznego pakietu w celu wyodrębnienia

nia wszystkich przestarzałych sygnatur metod. Alternatywna ścieżka, dostępna pod przyciskiem **Scan Project for Deprecated Usages**, inicjuje przeszukiwanie własnego kodu źródłowego użytkownika pod kątem odwołań do niezalecanych elementów.

Kluczowym elementem mechanizmu detekcji jest możliwość wyboru poziomu szczegółowości analizy projektu. Po uruchomieniu skanowania projektu system wyświetla dedykowane okno dialogowe z opcjami skanowania, które zaprezentowano na Rysunku 6.2.

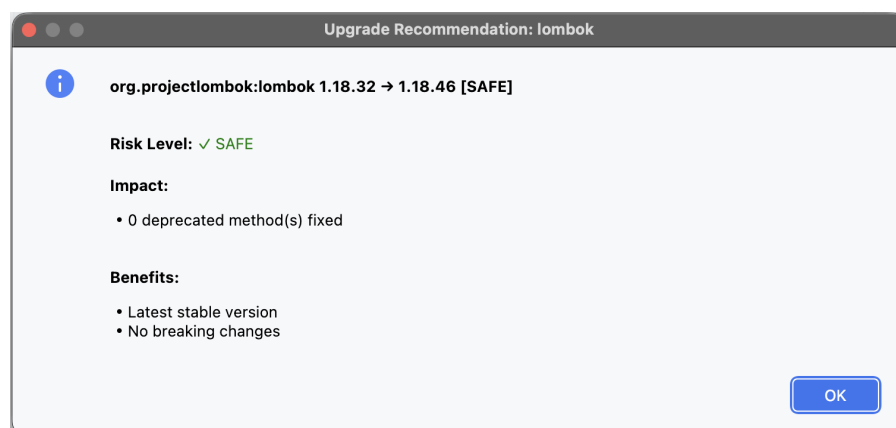


Rysunek 6.2: Okno dialogowe umożliwiające wybór trybu skanowania projektu. Źródło: opracowanie własne.

W tym momencie użytkownik podejmuje decyzję o sposobie pozyskiwania wiedzy technicznej, wybierając jeden z dwóch trybów pracy

- **Quick Scan** – tryb szybki, operujący wyłącznie na flagach kompilatora zapisanych w strukturach skompilowanych klas bibliotecznych, co pozwala na szybką identyfikację przestarzałych elementów, lecz często nie dostarcza szczegółowych informacji o możliwych zamiennikach,
- **Fetch Details** – tryb kompleksowy, w ramach którego system pobiera kompletne pakiety źródłowe wraz z powiązaną z nimi dokumentacją Javadoc, co umożliwia wyciągnięcie szczegółowych opisów oraz sugerowanych transformacji kodu bezpośrednio z Maven Repository.

Ostatnią funkcjonalnością dostępną z głównego interfejsu graficznego jest mechanizm weryfikacji aktualizacji, który został zilustrowany na Rysunku 6.3. Po wybraniu danej metody inżynier może użyć przycisku **Check Upgrades**, co pozwala na szybką ocenę stopnia krytyczności długu technologicznego oraz umożliwia sprawdzenie kosztu technicznego ewentualnego podbicia wersji biblioteki do najnowszej dostępnej odsłony.



Rysunek 6.3: Okno dialogowe przedstawiające raport złożoności aktualizacji zależności. Źródło: opracowanie własne.

6.2. Prezentacja wyników analizy

Po zakończeniu wybranego procesu skanowania zebrane dane trafiają bezpośrednio do odpowiednich sekcji w panelu bocznym wtyczki. Interfejs użytkownika organizuje te informacje w ramach dwóch niezależnych zakładkek, co pozwala na wyraźne rozdzielenie ogólnego skanu bibliotek zewnętrznych od bezpośrednich problemów zidentyfikowanych w plikach źródłowych dewelopera.

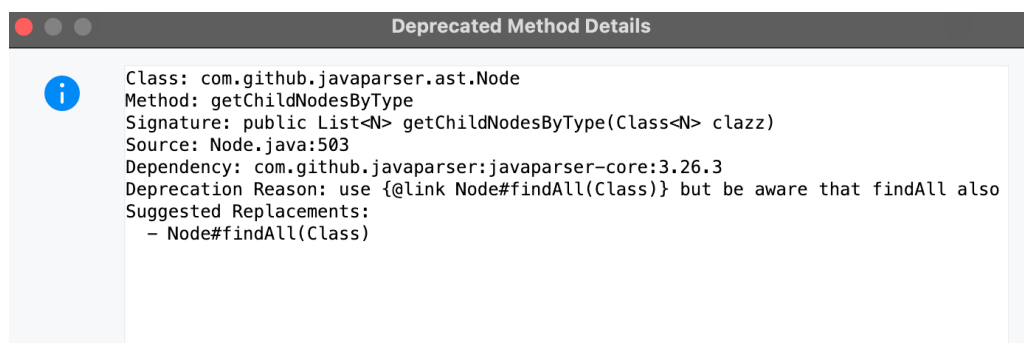
6.2.1. Weryfikacja wycofanych interfejsów w pakietach zewnętrznych

Pierwsza z dedykowanych sekcji analitycznych, dostępna pod zakładką **Deprecated Methods**, służy do prezentacji wszystkich niezalecanych elementów wykrytych w obrębie danej zależności. Widok ten jest zasilany po wywołaniu polecenia skanowania konkretnej biblioteki i generuje pełny wykaz struktur, które autorzy oprogramowania oznaczyli jako przestarzałe. Wygląd tego panelu z wypełnionymi danymi analitycznymi dla zależności `com.github.javaparser:3.26.3` przedstawia Rysunek 6.4.

Dependencies Deprecated Methods Method Usages					
Clear Results Export Results					
Class	Method	Reason	Suggested Replacem...	Dependency	Source File
com.github.javaparser....	isSpaceOrTab	Use {@link #isWhitesp...	#isWhitespaceButNotE...	com.github.javaparser:...	TokenTypes.java
com.github.javaparser....	getIndent	(@see Indentation.getl...	getIndentation, getInd...	com.github.javaparser:...	PrettyPrinterConfigura...
com.github.javaparser.printer.configuration.PrettyPrinterConfiguration	size	(@see Indentation.size())	getIndent, get, get	com.github.javaparser:...	PrettyPrinterConfigura...
com.github.javaparser....	setIndentSize	(@see Indentation.type)	None found	com.github.javaparser:...	PrettyPrinterConfigura...
com.github.javaparser....	getIndentType	(@see Indentation.typ...	getIndent, get, get	com.github.javaparser:...	PrettyPrinterConfigura...
com.github.javaparser....	setIndentType	(@see Indentation.size)	None found	com.github.javaparser:...	PrettyPrinterConfigura...
com.github.javaparser....	getTabWidth	(@see Indentation.size)	get, get	com.github.javaparser:...	PrettyPrinterConfigura...
com.github.javaparser....	setTabWidth	(@see Indentation.size)	None found	com.github.javaparser:...	PrettyPrinterConfigura...
com.github.javaparser....	output	No reason provided	output, outputDocume...	com.github.javaparser:...	XmlPrinter.java
com.github.javaparser....	getSource	use toString()	toString	com.github.javaparser:...	SourcePrinter.java
com.github.javaparser....	getSource	use toString()	toString	com.github.javaparser:...	PrettyPrintVisitor.java
com.github.javaparser....	findAncestor	This method is no long...	findAncestor(Predicat...	com.github.javaparser:...	HasParentNode.java
com.github.javaparser....	toCamelCase	use screamingToCame...	screamingToCamelCase	com.github.javaparser:...	Utils.java

Rysunek 6.4: Widok panelu wtyczki w zakładce **Deprecated Methods** dla `com.github.javaparser:3.26.3`. Źródło: opracowanie własne.

Tabela w tej sekcji dostarcza inżynierowi oprogramowania informacji na temat stanu technicznego importowanego pakietu, prezentując pełną strukturę klas, nazwy wycofanych metod, szczegółowe powody usunięcia elementów oraz oficjalne alternatywy. Interakcja z tabelą pozwala na kliknięcie wybranego wiersza, co skutkuje wyświetleniem dedykowanego okna szczegółów prezentującego zebrane dane w czytelniejszej dla programisty formie. Wygląd panelu z uszczegółowionymi informacjami o wybranej metodzie przedstawia Rysunek 6.5.



Rysunek 6.5: Szczegółowy widok informacji o przestarzałej metodzie po wybraniu elementu z tabeli dla metody `com.github.javaparser.ast.Node.getChildNodesByType`. Źródło: opracowanie własne.

Ważną cechą projektową omawianego modułu jest sposób agregacji danych. Inicjalizacja skanowania kolejnej biblioteki zewnętrznej nie nadpisuje aktualnego stanu widoku, lecz automatycznie dokleja nowe wykryte błędy do istniejącego zestawienia. Takie rozwiązanie umożliwia stopniowe budowanie zbiorczego raportu dla wielu komponentów bez utraty wcześniej zgromadzonych informacji.

Do zarządzania wygenerowaną zawartością służą przyciski funkcyjne umieszczone w obrębie panelu. Wybranie opcji **Clear Result** pozwala na zresetowanie widoku tabeli i usunięcie wszystkich wpisów z pamięci podręcznej wtyczki. Z kolei przycisk **Export Result** odpowiada za skopiowanie kompletnego raportu analitycznego bezpośrednio do schowka systemowego, co umożliwia deweloperowi szybkie przeniesienie surowych danych do zewnętrznych arkuszy.

6.2.2. Identyfikacja przestarzałego kodu w projekcie użytkownika

Druga zakładka, zatytułowana **Method Usages**, skupia się bezpośrednio na interakcji z kodem źródłowym samego projektu. W tym obszarze system gromadzi wyłącznie te przestarzałe metody, które zostały realnie wywołane przez programistę w plikach roboczych przestrzeni deweloperskiej. Wygląd tego panelu po użyciu opcji **Fetch Details** prezentuje Rysunek 6.6.

Class	Method	Suggested Replacement	Used In Class	Used In Method	File	Line
com.github.javapars...	getChildNodesByType	Node#findAll(Class)	LegacyApiUsage	demonstrateGet...	LegacyApiUsage.java	25
com.github.javapars...	getComments	getAllComments()	LegacyApiUsage	demonstrateGet...	LegacyApiUsage.java	31
com.github.javapars...	asInt	IntegerLiteralExpr#asNumber()	LegacyApiUsage	demonstrateInte...	LegacyApiUsage.java	38
com.github.javapars...	asLong	LongLiteralExpr#asNumber()	LegacyApiUsage	demonstrateLon...	LegacyApiUsage.java	46
com.github.javapars...	getIndentSize	getIndent, get, get	LegacyApiUsage	demonstrateInde...	LegacyApiUsage.java	54
com.github.javapars...	getConfiguration	getParserConfiguration()	Hephaestus	main	Hephaestus.java	27
java.util.Date	getYear	None found	DeprecatedMeth...	mainMethods	DeprecatedMethodsUsag...	11
java.net.URL	URL	None found	DeprecatedMeth...	downloadFile	DeprecatedMethodFinder...	87
pl.pjatk.testfiles.Dep...	oldMethod	None found	DeprecatedMeth...	main	DeprecatedMethodsUsag...	6

Rysunek 6.6: Widok panelu wtyczki w zakładce **Method Usages** po użyciu opcji **Fetch Details**. Źródło: opracowanie własne.

Główny pasek narzędziowy tej sekcji udostępnia zestaw przycisków sterujących procesem zarządzania przestarzałymi metodami użytymi w kodzie. Przycisk **Clear Results** pozwala na wyczyszczenie bieżącej sesji analitycznej, natomiast **Navigate to Selected** przenosi kursor edytora bezpośrednio do wybranego naruszenia. Interakcję graficzną wspierają funkcje **Highlight All in Editor** oraz **Clear Highlights**, które odpowiadają za sterowanie kolorowymi wyróżnieniami wewnątrz plików źródłowych. Całość zestawienia może zostać zarchiwizowana za pomocą przycisku **Export Results**, który kopiuje surowe dane do schowka systemowego.

Kluczowym elementem sterującym procesem modernizacji jest przycisk **Refactor Selected**. Został on zaprojektowany w formie menu rozwijanego, które agreguje dwie niezależne ścieżki transformacji kodu. Pierwsza z dostępnych opcji to **AI Refactor Selected**, która deleguje zadanie przebudowy kodu do modelu językowego. Druga funkcja pod nazwą **Build-in Refactor Selected** odpowiada za realizację zmian na podstawie reguł odnalezionych w dokumentacji Javadoc.

Centralną część omawianego widoku zajmuje tabela wyników, która dostarcza informacji o zidentyfikowanym długu technologicznym. Struktura danych w tym oknie została podzielona na siedem pól

- **Class** – nazwa przestarzałej klasy pochodzącej z biblioteki zewnętrznej,
- **Method** – nazwa konkretnej metody wycofanej z użytku,
- **Suggested Replacement** – treść wskazówek migracyjnych pobranych z dokumentacji technicznej,

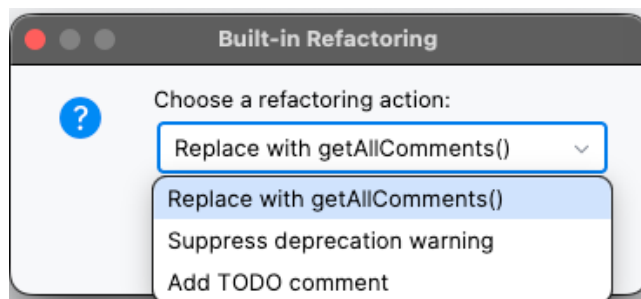
- **Used in Class** – nazwa klasy w projekcie użytkownika, w której wykryto problem,
- **Used in Method** – nazwa metody dewelopera zawierającej nieprawidłowe wywołanie,
- **File** – dokładna nazwa pliku źródłowego podlegającego analizie,
- **Line** – numer linii, w której znajduje się metoda wymagająca modernizacji.

Taki podział danych pozwala na precyzyjne planowanie kolejnych kroków refaktoryzacyjnych. Interaktywna lista stanowi pomost między analizą a aktywnym nanoszeniem poprawek, co znacząco skraca czas potrzebny na aktualizację zależności w rozbudowanych systemach informatycznych.

6.3. Interaktywne mechanizmy naprawcze

Po zidentyfikowaniu długu technologicznego i wybraniu odpowiedniego wiersza w tabeli wyników użytkownik zyskuje możliwość automatycznego usunięcia wykrytych błędów składniowych. Transformacja kodu jest inicjowana bezpośrednio z poziomu paska narzędziowego w zakładce panelu za pomocą rozwijanego przycisku **Refactor Selected**. System udostępnia cztery komplementarne przypadki użycia zorganizowane w ramach dwóch głównych modułów wykonawczych.

Pierwszy moduł uruchamiany za pomocą opcji **Build-in Refactor Selected** skupia się na akcjach deterministycznych. Kluczowym etapem konfiguracji naprawy jest wybór konkretnego wariantu transformacji wewnątrz menu modułu. Sposób prezentacji tych opcji oraz ich strukturę w interfejsie graficznym ilustruje Rysunek 6.7.



Rysunek 6.7: Menu wyboru konkretnej strategii naprawczej w ramach modułu **Build-in Refactor Selected**. Źródło: opracowanie własne.

Gdy deweloper aktywuje sekcję rozwiązań systemowych, program wyświetla trzy dostępne ścieżki operacyjne. Pierwszą z nich jest opcja **Refactor Selected**, która odpowiada za bezpośrednią podmianę kodu. Drugi wariant pozwala na wybranie funkcji **Add TODO comment**, co skutkuje wstrzyknięciem znacznika planistycznego nad wadliwą linią. Ostatnią możliwością jest wybranie komendy **Add SuppressWarnings annotation**, która służy do wyciszenia błędów kompilacji bez modyfikacji samej logiki wywołania.

Kompleksowe zestawienie porównawcze prezentujące stan wejściowy programu oraz efekty wybrania każdej z wymienionych wyżej strategii w układzie macierzowym przedstawia Rysunek 6.8.

Wariant A ilustruje stan pierwotny, w którym widoczne jest bezpośrednie odwołanie do przestarzałego elementu. W ramach akcji zaprezentowanej jako Wariant B system dokonał podmiany metody na nowy odpowiednik. Wariant C przedstawia wstrzyknięty znacznik tekstowy, natomiast Wariant D obrazuje dodaną instrukcję ignorującą ostrzeżenia kompilatora dla danego bloku kodu.

```
public void demoGetComments(CompilationUnit cu) { 1 usage
    // CompilationUnit.getComments() is deprecated
    cu.getComments();
}
```

(A) Oryginalny kod źródłowy przed transformacją

```
public void demoGetComments(CompilationUnit cu) { 1 usage
    cu.getAllComments();
}
```

(B) Automatyczna podmiana metody

```
public void demoGetComments(CompilationUnit cu) { 1 usage
    // TODO: Replace deprecated method call
    cu.getComments();
}
```

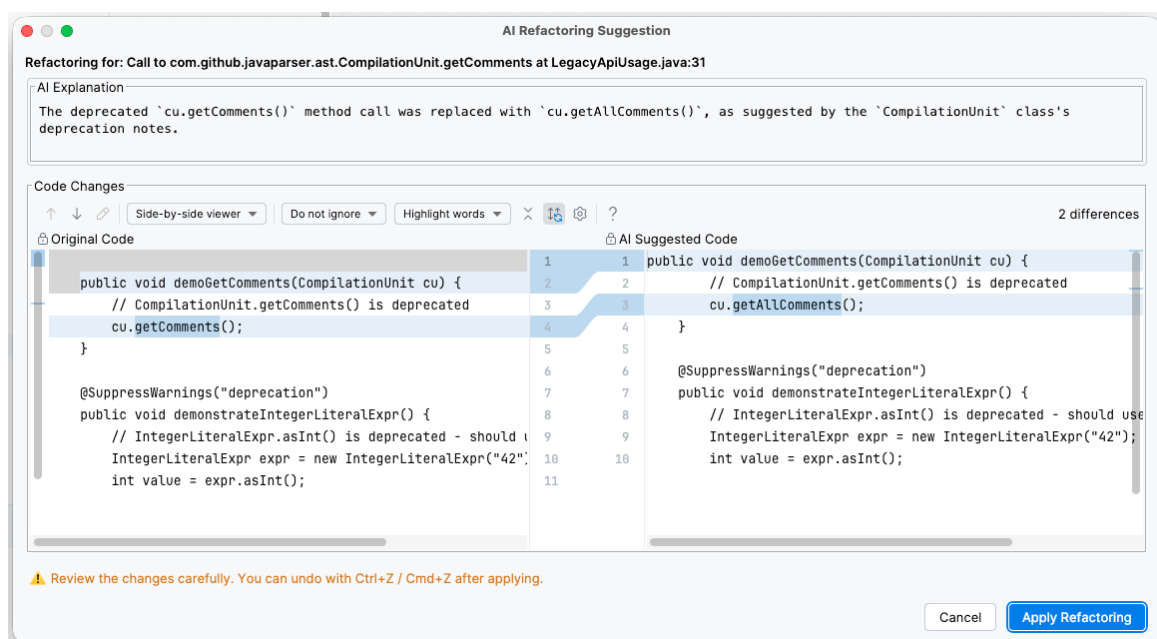
(C) Wygenerowany komentarz TODO

```
@SuppressWarnings("deprecation") 1 usage
public void demoGetComments(CompilationUnit cu) {
    cu.getComments();
}
```

(D) Wstrzyknięta adnotacja wyciszająca błędy

Rysunek 6.8: Zestawienie porównawcze struktury kodu źródłowego obejmujące stan pierwotny oraz trzy alternatywne rezultaty działania wbudowanego silnika regułowego. Źródło: opracowanie własne.

Dla skomplikowanych naruszeń architektonicznych przeznaczona jest ścieżka inteligentnej refaktoryzacji dostępna pod opcją **AI Refactor Selected**. Stanowi ona czwarty fundamentalny przypadek użycia systemu, w którym model językowy rekonstruuje logikę wywołania na podstawie przekazanego kodu źródłowego oraz kontekstu. Kluczową warstwą bezpieczeństwa chroniącą projekt przed ewentualnymi błędami składniowymi sztucznej inteligencji jest interaktywny widok różnicujący zaprezentowany na Rysunku 6.9.



Rysunek 6.9: Interaktywne okno porównawcze prezentujące kod przed i po transformacji AI. Źródło: opracowanie własne.

Okno różnicowe daje programiście pełną kontrolę nad ostatecznym kształtem kodu poprzez graficzne zestawienie wprowadzanych zmian. W tym miejscu inżynier oprogramowania podejmuje ostateczną decyzję wykonawczą o naniesieniu poprawek. Kliknięcie przycisku zatwierdzenia powoduje uruchomienie transakcji zapisu i trwale naniesienie zmian w strukturze pliku źródłowego. Wybranie opcji anulowania skutkuje natychmiastowym przerwaniem całej procedury oraz bezpiecznym powrotem do edytora bez modyfikowania jakichkolwiek zasobów w projekcie dewelopera.

6.4. Analiza z wykorzystaniem interfejsu wiersza poleceń

Komponent konsolowy w formie niezależnego pliku wykonywalnego o nazwie `hephaestus-cli.jar` umożliwia uruchamianie procesów analitycznych w środowiskach całkowicie pozbawionych interfejsu graficznego. Narzędzie to zostało zaprojektowane z myślą o automatyzacji audytu kodu oraz łatwej integracji z potokami CI/CD. Działanie programu opiera się na wywoływaniu pliku narzędziowego z odpowiednimi argumentami tekstowymi oraz flagami konfiguracyjnymi, które determinują tryb pracy silnika diagnostycznego.

6.4.1. Skanowanie pojedynczej zależności zewnętrznej

Polecenie `scan` odpowiada za izolowaną weryfikację wskazanej biblioteki zdefiniowanej przez współrzędne repozytorium Maven. Narzędzie przyjmuje jako argumenty kolejno identyfikator grupy oraz identyfikator artefaktu a także numer wersji pakietu. Podsystem pobiera dane wybranej zależności, a następnie generuje w oknie konsoli pełną strukturę przestarzałych sygnatur metod. Ogólny szablon składniowy tego polecenia prezentuje listing 6.1.

Listing 6.1: Ogólna składnia polecenia `scan`. Źródło: opracowanie własne.

```
1 java -jar hephaestus-cli.jar scan <groupId> <artifactId> <version>
```

Wyszukiwanie przestarzałych metod dla konkretnego przypadku realizuje listing 6.2, w którym analizie poddano zewnętrzną bibliotekę `javaparser-core` w wersji 3.26.3.

Listing 6.2: Wywołanie polecenia `scan` dla zależności `javaparser-core`. Źródło: opracowanie własne.

```
1 java -jar hephaestus-cli.jar scan com.github.javaparser javaparser-core 3.26.3
```

Rzeczywisty rezultat tekstowy wygenerowany na ekranie terminala dla tego konkretnego przypadku analizy ilustruje listing 6.3. W celu czytelniejszego przedstawienia wyników punkty od 3 do 40 zostały pominięte.

Listing 6.3: Wynik polecenia `scan` prezentujący wycofane metody. Źródło: opracowanie własne.

```
1 =====
2                               SCAN RESULTS
3                               =====
4 Found 42 deprecated method(s):
5
6 1. com.github.javaparser.printer.configuration.PrettyPrinterConfiguration.getIndent
7   Source: PrettyPrinterConfiguration.java:68
8   Reason: (@see Indentation.getIndent())
9   Suggested replacements: getIndentation, getIndentSize, getIndentType, get, get
10  Method signature: public String getIndent()
11
12 2. com.github.javaparser.printer.configuration.PrettyPrinterConfiguration.
13   getIndentSize
14   Source: PrettyPrinterConfiguration.java:77
15   Reason: (@see Indentation.size)
16   Suggested replacements: getIndent, get, get
17   Method signature: public int getIndentSize()
18 [...]
19
20 41. com.github.javaparser.StaticJavaParser.parseResource
21   Source: StaticJavaParser.java:224
22   Reason: set the encoding in the {@link ParserConfiguration}
23   Suggested replacements: ParserConfiguration
```

```

24     Method signature: public static CompilationUnit parseResource(
25         @NotNull final ClassLoader classLoader,
26         @NotNull final String path,
27         @NotNull Charset encoding) throws IOException
28
29 42. com.github.javaparser.StaticJavaParser.handleResult
30     Source: StaticJavaParser.java:546
31     Method signature: private static T handleResult(ParseResult<T> result)
32
33 =====
34 Summary: 42 deprecated methods found
35 - 38 with suggested replacements
36 - 4 without replacements
37 =====

```

6.4.2. Porównywanie wersji zależności

Ocenę stopnia dezaktualizacji biblioteki ułatwia komenda `compare`, która służy do zestawiania ze sobą dwóch różnych wydań tego samego pakietu zewnętrznego. Program analizuje rejestry wycofanych metod dla wersji źródłowej oraz wersji docelowej a następnie generuje raport prezentujący listę zmian w obrębie przestarzałego interfejsu. Sposób wywołania narzędzia prezentuje listing 6.4.

Listing 6.4: Ogólna składnia polecenia `compare`. Źródło: opracowanie własne.

```

1 java -jar hephaestus-cli.jar compare <groupId> <artifactId> <fromVersion> <toVersion>

```

Wywołanie procesu porównania dla konkretnego przypadku ilustruje listing 6.5, w którym zestawiono wydanie biblioteki testowej JUnit w wersji 4.12 z nowszą edycją oznaczoną numerem 4.13.2.

Listing 6.5: Wywołanie polecenia `compare` dla biblioteki JUnit. Źródło: opracowanie własne.

```

1 java -jar hephaestus-cli.jar compare junit junit 4.12 4.13.2

```

Raport końcowy zestawiający ogólny wpływ aktualizacji na projekt dla tego konkretnego przypadku zawiera listing 6.6.

Listing 6.6: Wynik wywołanie polecenia `compare` dla biblioteki JUnit. Źródło: opracowanie własne.

```

1 =====
2             VERSION COMPARISON REPORT
3 =====
4 Old version: 4.12 (21 deprecated methods)
5 New version: 4.13.2 (25 deprecated methods)
6
7 NEW DEPRECATIONS (4 methods)
8 These methods became deprecated in version 4.13.2:
9
10 1. org.junit.Assert.assertThat
11    Location: Assert.java:928
12    Reason: use {@code org.hamcrest.MatcherAssert.assertThat()}
13    -> Migrate to: org.hamcrest.MatcherAssert.assertThat()
14
15 2. org.junit.Assert.assertThat
16    Location: Assert.java:961
17    Reason: use {@code org.hamcrest.MatcherAssert.assertThat()}
18    -> Migrate to: org.hamcrest.MatcherAssert.assertThat()
19
20 3. org.junit.rules.ExpectedException.none
21    Location: ExpectedException.java:121

```

```

22 Reason: Since 4.13
23 {@link org.junit.Assert#assertThrows(Class,
24 org.junit.function.ThrowingRunnable) Assert.assertThrows}
25 can be used to verify that your code throws a specific
26 exception.
27 -> Migrate to: org.junit.Assert#assertThrows(Class,
28 org.junit.function.ThrowingRunnable) Assert.assertThrows
29
30 4. org.junit.runners.ParentRunner.createTestClass
31 Location: ParentRunner.java:109
32 Reason: Please
33 -> Migrate to: ParentRunner(org.junit.runners.model.TestClass)
34
35 STILL DEPRECATED (21 methods)
36 These methods remain deprecated in version 4.13.2:
37
38 - org.junit.experimental.max.MaxCore.forFolder
39 - org.junit.internal.JUnitSystem.exit
40 - org.junit.internal.RealSystem.exit
41 - org.junit.Assert.assertEquals
42 - org.junit.Assert.assertEquals
43 - org.junit.Assert.assertEquals
44 - org.junit.Assert.assertEquals
45 - org.junit.rules.ExpectedException.handleAssertionErrors
46 - org.junit.rules.ExpectedException.handleAssumptionViolatedExceptions
47 - org.junit.rules.TestWatcher.skipped
48 - org.junit.runners.model.FrameworkMethod.producesType
49 - org.junit.runners.BlockJUnit4ClassRunner.validateInstanceMethods
50 - org.junit.runners.BlockJUnit4ClassRunner.withPotentialTimeout
51 - org.junit.matchers.JUnitMatchers.hasItem
52 - org.junit.matchers.JUnitMatchers.hasItem
53 - org.junit.matchers.JUnitMatchers.hasItems
54 - org.junit.matchers.JUnitMatchers.hasItems
55 - org.junit.matchers.JUnitMatchers.everyItem
56 - org.junit.matchers.JUnitMatchers.containsString
57 - org.junit.matchers.JUnitMatchers.both
58 - org.junit.matchers.JUnitMatchers.either
59
60 =====
61 UPGRADE IMPACT SUMMARY:
62
63 Total deprecation changes: 4
64 + 4 new deprecations
65 - 0 removed deprecations
66 = 21 still deprecated
67
68 RECOMMENDATION:
69 Before upgrading, check if your code uses any of the newly
70 deprecated methods listed above and plan refactoring accordingly.
71 =====

```

6.4.3. Masowy audyt lokalnego projektu

Do przeszukiwania plików źródłowych w przestrzeni roboczej dewelopera przeznaczone jest polecenie `scan-project`, które jako główny parametr przyjmuje ścieżkę do katalogu z projektem. Tryb ten obsługuje modyfikatory działania w postaci opcjonalnych przełączników konfiguracyjnych. Deweloper ma do dyspozycji dwa parametry sterujące

- `--full` – aktywuje pełną analizę opartą na pobieraniu pakietów źródłowych oraz parsowaniu dokumentacji Javadoc,

- `--json` – zmienia domyślny format wyjściowy raportu z tekstu na ustrukturyzowany obiekt wymiany danych JSON przeznaczony dla zewnętrznych systemów raportowania błędów.

Standardowe uruchomienie podstawowego audytu kodu bez stosowania dodatkowych modyfikatorów realizuje listing 6.7.

Listing 6.7: Podstawowe wywołanie `scan-project` bez flag dodatkowych. Źródło: opracowanie własne.

```
1 java -jar hephaestus-cli.jar scan-project /path/to/project/test-project
```

Odpowiedź systemu zawierająca uproszczony wykaz błędów znajduje się na listingu 6.8. W celu czytelniejszego przedstawienia wyników punkty od 3 do 5 zostały pominięte.

Listing 6.8: Wynik tekstowy polecenia `scan-project` w trybie szybkim. Źródło: opracowanie własne.

```
1 =====
2 PROJECT DEPRECATION SCAN
3 =====
4 Project: /path/to/project/test-project
5 Scan mode: QUICK (bytecode only)
6
7 Scanning compiled bytecode for deprecated method usages...
8
9 =====
10 QUICK SCAN RESULTS
11 =====
12 Found 7 deprecated method usage(s):
13
14 1. com.github.javaparser.StaticJavaParser.getConfiguratio()
15    Lcom/github/javaparser/ParserConfiguration;
16    Used in 1 location(s):
17      - pl.pjatk.Hephaestus.main -> Hephaestus.java:27
18
19 2. com.github.javaparser.printer.configuration.
20    PrettyPrinterConfiguration.getIndentSize()I
21    Used in 1 location(s):
22      - pl.pjatk.testfiles.LegacyApiUsage.
23        demonstrateIndentConfiguration -> LegacyApiUsage.java:52
24
25 [...]
26
27 6. com.github.javaparser.ast.CompilationUnit.
28    getChildNodesByType(Ljava/lang/Class;)Ljava/util/List;
29    Used in 1 location(s):
30      - pl.pjatk.testfiles.LegacyApiUsage.
31        demonstrateGetChildNodesByType -> LegacyApiUsage.java:24
32
33 7. com.github.javaparser.ast.expr.LongLiteralExpr.asLong()J
34    Used in 1 location(s):
35      - pl.pjatk.testfiles.LegacyApiUsage.
36        demonstrateLongLiteralExpr -> LegacyApiUsage.java:44
37
38 =====
39 SUMMARY:
40   Total usages found: 7
41   Unique deprecated methods: 7
42
43 TIP: Use --full flag for detailed deprecation information
44      (requires fetching Maven sources)
45 =====
```

Wymuszenie kompleksowej weryfikacji połączonej z pełną ekstrakcją zaleceń migracyjnych z dokumentacji technicznej wykonuje listing 6.9.

Listing 6.9: Wywołanie pełnego skanowania projektu z flagą `--full`. Źródło: opracowanie własne.

```
1 java -jar hephaestus-cli.jar scan-project /path/to/project/test-project --full
```

Dodatkowy element, otrzymany poprzez dodanie flagi `--full`, zawierający powody deprecji oraz sugerowane zamienniki wyciągnięte z dokumentacji prezentuje listing 6.10.

Listing 6.10: Dodatkowy element raportu otrzymanego z wykorzystaniem polecenia `scan-project --full`. Źródło: opracowanie własne.

```
1 =====
2             FETCHING DETAILED DEPRECATION INFO
3 =====
4
5 Method: com.github.javaparser.StaticJavaParser.getConfiguration
6 Reason: use {@link #getParserConfiguration()} instead
7 Upgrade to: getParserConfiguration()
8 Source: StaticJavaParser.java:61
9
10 Method: com.github.javaparser.printer.configuration.
11         PrettyPrinterConfiguration.getIndentSize
12 Reason: (@see Indentation.size)
13 Upgrade to: getIndent, get, get
14 Source: PrettyPrinterConfiguration.java:77
15
16 Method: com.github.javaparser.ast.CompilationUnit.getComments
17 Reason: getComments was a too generic name and it could be
18         confused with getComment or getAllContainedComments
19 Upgrade to: getAllComments()
20 Source: CompilationUnit.java:179
21
22 Method: com.github.javaparser.ast.expr.IntegerLiteralExpr.asInt
23 Reason: This function has issues with corner cases,
24         such as 2147483648, so please
25 Upgrade to: IntegerLiteralExpr#asNumber()
26 Source: IntegerLiteralExpr.java:100
27
28 Method: com.github.javaparser.ast.expr.LongLiteralExpr.asLong
29 Reason: This function has issues with corner cases,
30         such as 9223372036854775808L, so please
31 Upgrade to: LongLiteralExpr#asNumber()
32 Source: LongLiteralExpr.java:102
33
34 =====
```

Jednoczesne wygenerowanie strukturalnego pliku wynikowego oraz przeprowadzenie pełnego skanowania zasobów projektowych zapewnia połączenie obu przełączników konfiguracyjnych. Komendę realizującą takie wywołanie przedstawia listing 6.11.

Listing 6.11: Wywołanie skanowania z formatowaniem JSON oraz pełnym opisem. Źródło: opracowanie własne.

```
1 java -jar hephaestus-cli.jar scan-project /path/to/project/test-project --json --full
```

Wynik takiego wywołania sformatowany w standardzie wymiany danych przedstawia listing 6.12. W celu czytelniejszego przedstawienia wyników elementy od 2 do 6 zostały pominięte.

Listing 6.12: Rezultat skanowania projektu z użyciem flag `--json` oraz `--full`. Źródło: opracowanie własne.

```
1 {
2   "projectPath": "/path/to/project/test-project",
3   "totalUsages": 7,
4   "uniqueMethods": 7,
5   "deprecatedMethods": [
6     {
7       "className": "com.github.javaparser.StaticJavaParser",
8       "methodName": "getConfiguration",
9       "descriptor": "()Lcom/github/javaparser/ParserConfiguration;",
10      "usageCount": 1,
11      "reason": "use {@link #getParserConfiguration()} instead",
12      "replacements": ["getParserConfiguration()"],
13      "usages": [
14        {
15          "callerClass": "pl.pjatk.Hephaestus",
16          "callerMethod": "main",
17          "sourceFile": "Hephaestus.java",
18          "lineNumber": 27
19        }
20      ]
21    },
22    [...]
23  ]
24  {
25    "className": "com.github.javaparser.ast.expr.LongLiteralExpr",
26    "methodName": "asLong",
27    "descriptor": "()J",
28    "usageCount": 1,
29    "reason": "This function has issues with corner cases, such as
30      9223372036854775808L, so please",
31    "replacements": ["LongLiteralExpr#asNumber()"],
32    "usages": [
33      {
34        "callerClass": "pl.pjatk.testfiles.LegacyApiUsage",
35        "callerMethod": "demonstrateLongLiteralExpr",
36        "sourceFile": "LegacyApiUsage.java",
37        "lineNumber": 44
38      }
39    ]
40  }
41 ]
42 }
```

6.4.4. Wspomaganie użytkownika w trybie interaktywnym

Ostatnią funkcjonalnością modułu wiersza poleceń jest tryb interaktywny który został zaprojektowany z myślą o deweloperach preferujących sterowane kreatory konfiguracji. Uruchomienie narzędzia konsolowego bez przekazywania jakichkolwiek argumentów startowych skutkuje aktywowaniem dedykowanego menu pomocniczego. Sposób inicjalizacji trybu asysty przedstawia listing 6.13.

Listing 6.13: Inicjalizacja modułu konsolowego w trybie interaktywnym. Źródło: opracowanie własne.

```
1 java -jar hephaestus-cli.jar
```

Widok powitalnego menu sterowanego za pomocą wprowadzania liczb ilustruje listing 6.14.

Listing 6.14: Rezultat uruchomienia modułu konsolowego w trybie interaktywnym. Źródło: opracowanie własne.

```
1 +-----+
2 |           Hephaestus CLI           |
3 |           Deprecated Method Finder   |
4 +-----+
5
6 Welcome to Hephaestus Interactive Mode!
7 This tool helps you find deprecated methods in Maven dependencies.
8
9 Options:
10 1. Scan a Maven dependency
11 2. Compare two versions
12 3. Scan project for deprecated usages
13 4. Help
14 5. Exit
15 Please select an option (1-5):
```

7. Podsumowanie i wnioski końcowe

Niniejszy rozdział stanowi podsumowanie wszystkich wyników pracy. Zawiera on szczegółową analizę problemów napotkanych podczas implementacji prototypu oraz ocenę mocnych i słabych stron gotowego narzędzia. W podsumowaniu wskazano również potencjalne ścieżki ewolucji oprogramowania oraz sformułowano wnioski ogólne wynikające z przeprowadzonego procesu badawczego.

7.1. Trudności napotkane podczas realizacji projektu

Proces projektowania oraz implementacji wtyczki do środowiska **IntelliJ IDEA** wiązał się z koniecznością rozwiązania specyficznych problemów technicznych. Analiza poszczególnych etapów prac pozwoliła na wyodrębnienie trzech kluczowych wyzwań o charakterze programistycznym, do których należały przedstawione poniżej zagadnienia

- **Pobieranie oraz analiza zewnętrznych plików zawierających informacje dokumentacyjne z poziomu wtyczki** – głównym wyzwaniem strukturalnym był brak lokalnego dostępu do pełnych opisów metod dostarczanych przez biblioteki zewnętrzne w trakcie skanowania projektu użytkownika. Problem ten rozwiązano poprzez zaimplementowanie mechanizmu pobierania właściwych archiwów bezpośrednio z centralnego repozytorium **Maven**, które są następnie automatycznie zapisywane w tymczasowym katalogu systemowym na czas wykonywania operacji analitycznych.
- **Wyszukiwanie informacji o sugerowanych zamiennikach przestarzałych metod w strukturach komentarzy Javadoc** – niejednolity format wpisów dokumentacyjnych utrudniał jednoznaczne i automatyczne wyciąganie referencji do nowych interfejsów programistycznych. Wyzwanie to zostało zrealizowane poprzez wdrożenie zestawu sztywnych reguł wyszukiwania danych opartych na analizie słów kluczowych oraz bezpośrednich odnośnikach do innych metod, uzupełniając algorytm o procesy identyfikacji funkcji przeciążonych. Dla scenariusza, w którym system nie odnajdzie żadnych rekomendacji migracyjnych, aplikacja umożliwia wstrzyknięciu znacznika **TODO** do późniejszej weryfikacji bądź zaaplikowaniu adnotacji tłumiącej ostrzeżenia kompilatora.
- **Brak dostępności mechanizmów platformy programistycznej w module konsolowym** – zasadnicza trudność polegała na braku możliwości przeprowadzenia analizy struktur językowych w taki sam sposób jak wewnątrz wtyczki graficznej. Narzędzie **CLI** funkcjonuje poza środowiskiem **IntelliJ IDEA**, przez co jest całkowicie pozbawione dostępu do interfejsów programistycznych oraz uniwersalnego modelu reprezentacji kodu **PSI**. Aby rozwiązać ten problem i umożliwić niezależne działanie programu, konieczne było wdrożenie zupełnie odrębnej ścieżki analitycznej opartej na niskopoziomowej bibliotece **ASM**, która realizuje skanowanie bezpośrednio na skompilowanym kodzie bajtowym plików klas.

7.2. Mocne i słabe strony rozwiązania

Opracowany prototyp systemu poddano wszechstronnej ocenie pod kątem jego użyteczności. Synteza zebranych wniosków pozwoliła na obiektywne wskazanie kluczowych zalet oraz ograniczeń funkcjonalnych, które wynikają bezpośrednio z podjętych decyzji projektowych oraz technologicznych na etapie implementacji.

7.2.1. Mocne strony systemu

Do najważniejszych zalet opracowanego rozwiązania zaliczają się następujące cechy:

- **Architektura dualna łącząca środowisko graficzne z wierszem poleceń** – program może być uruchamiany w sposób interaktywny jako wtyczka do aplikacji IntelliJ IDEA, co doskonale wspiera codzienną pracę deweloperską, a także w trybie wsadowym przy użyciu modułu CLI, co pozwala na łatwą automatyzację audytów w potokach ciągłej integracji oprogramowania.
- **Bezpieczna i w pełni kontrolowana modyfikacja kodu źródłowego** – wszystkie transformacje strukturalne wtyczki są realizowane wewnątrz transakcji zapisu, co gwarantuje pełną obsługę operacji cofania oraz ponawiania zmian, a dodatkowy podgląd widoku różnicowego przed ostatecznym zatwierdzeniem eliminuje ryzyko wprowadzenia nieodwracalnych uszkodzeń do repozytorium.
- **Strukturalna integracja wyników dzięki eksportowi do formatu JSON** – implementacja obsługi ustrukturyzowanego formatu wymiany danych w module konsolowym umożliwia automatyczne przekazywanie rezultatów masowego skanowania do zewnętrznych systemów agregacji błędów, pulpitu menedżerskich oraz zautomatyzowanych potoków weryfikacji jakości oprogramowania.
- **Automatyczne generowanie akcji alternatywnych w obliczu braku zaleceń migracyjnych** – w sytuacji gdy analizowana dokumentacja zewnętrzna nie wskazuje bezpośredniego zamiennika dla wycofywanej metody system oferuje inteligentne scenariusze awaryjne w postaci automatycznego wstawienia komentarza typu TODO do późniejszej rewizji bądź zaaplikowania adnotacji @SuppressWarnings usuwającej ostrzeżenia kompilatora o deprecjacji kodu.
- **Skalowanie doświadczenia użytkownika podczas operacji długotrwałych** – procesy silnie obciążające zasoby sieciowe lub procesorowe są uruchamiane asynchronicznie poza głównym wątkiem graficznym edytora, a bieżący stan prac jest obrazowany za pomocą dedykowanych pasków postępu, co skutecznie zapobiega zjawisku zamrażania interfejsu aplikacji.
- **Wymierna oszczędność czasu inżyniera oprogramowania** – automatyczne wykrywanie odwołań do przestarzałych interfejsów programistycznych połączone z dostarczaniem kontekstowych sugestii i półautomatyczną refaktoryzacją pozwala na radykalne skrócenie czasu potrzebnego na wykonanie audytów bezpieczeństwa oraz ręcznych poprawek kodu.

7.2.2. Słabe strony systemu

Pomimo licznych zalet prototyp posiada pewne wady oraz limity technologiczne, które stanowią naturalny obszar do dalszych prac optymalizacyjnych

- **Niedojrzałe mechanizmy zarządzania sekretami i kluczami dostępowymi** – klucz uwierzytelniający serwisów sztucznej inteligencji nie jest w obecnej wersji obsługiwany przez dedykowany i bezpieczny sejf haseł systemu operacyjnego, co stanowi ograniczenie w obszarze bezpieczeństwa operacyjnego.

- **Silna zależność jakości generowanych rekomendacji od stanu dokumentacji wejściowej** – w sytuacji gdy autorzy biblioteki zewnętrznej nie umieścili w plikach źródłowych precyzyjnych tagów Javadoc lub adnotacji deprecji, system traci zdolność do automatycznego wskazywania właściwych zamienników architektonicznych.
- **Ograniczenia funkcjonalne skanera kodu bajtowego** – tryb oparty na bibliotece ASM wymaga uprzedniego poprawnego skompilowania projektu przez użytkownika i wykazuje tendencję do pomijania specyficznych struktur, do których należą skomplikowane klasy wewnętrzne, przez co nie stanowi on pełnego substytutu dla analizy kodu źródłowego.
- **Nierównomierne wsparcie dla popularnych systemów budowania projektów** – procesy rozwiązywania zależności oraz pobierania metadanych zostały w pełni zoptymalizowane pod kątem tylko ekosystemu Maven, co ogranicza uniwersalność narzędzia.
- **Niedeterministyczny charakter generowania sugestii naprawczych przez modele językowe** – probabilistyczna natura działania algorytmów generatywnej sztucznej inteligencji sprawia, że ten sam fragment kodu wejściowego może skutkować otrzymaniem zróżnicowanych rezultatów w kolejnych iteracjach skanowania. Cecha ta wprowadza element nieprzewidywalności do procesów automatycznej refaktoryzacji, ponieważ model LLM może w niespójny sposób dobierać zamienniki przestarzałych metod bądź sporadycznie generować propozycje zawierające błędy logiczne i składniowe, co wymusza ciągłą i bardzo uważną weryfikację każdej rekomendacji bezpośrednio przez inżyniera oprogramowania.

7.3. Kierunki dalszego rozwoju projektu

Opracowany prototyp wtyczki w obecnej formie stanowi stabilną oraz funkcjonalną bazę wyjściową do dalszych prac badawczo-rozwojowych. Pomimo pełnego zrealizowania wszystkich pierwotnych założeń analiza działania systemu w warunkach testowych pozwoliła na zidentyfikowanie kilku kluczowych obszarów, które warto rozwinąć w przyszłości.

Pierwszym i najbardziej naturalnym krokiem ewolucyjnym jest rozszerzenie obsługi mechanizmów zarządzania projektami o system **Gradle**. Obecna wersja systemu współpracuje wyłącznie ze strukturą plików konfiguracyjnych narzędzia **Maven**. Wdrożenie dedykowanego parsera dla skryptów budowania pisanych w językach **Groovy** oraz **Kotlin** znacznie zwiększyłoby uniwersalność aplikacji, umożliwiając jej efektywne wykorzystanie w projektach mobilnych oraz w nowoczesnych architekturach mikroserwisowych.

Kolejnym kluczowym usprawnieniem jest przebudowa podsystemu zarządzania modułem sztucznej inteligencji. W obecnej fazie rozwoju prototypu dostosowanie parametrów pracy modelu sztucznej inteligencji wymaga od użytkownika ręcznej edycji plików konfiguracyjnych aplikacji, co stanowi istotne utrudnienie. W ramach dalszego rozwoju planuje się zaimplementowanie dedykowanego graficznego panelu ustawień zintegrowanego bezpośrednio z oknem środowiska programistycznego. Interfejs ten pozwoli na uproszczoną i w pełni wizualną konfigurację silnika sztucznej inteligencji, eliminując potrzebę bezpośredniej modyfikacji zasobów strukturalnych programu. Jako Kluczowy element nowej architektury proponuje się dedykowany moduł bezpiecznego przechowywania kluczy dostępowych API, który zostanie zintegrowany z natywnym mechanizmem zarządzania sekretami platformy **IntelliJ**. Rozwiązanie to wyeliminuje dotychczasowe podatności operacyjne, zapewniając bezpieczne szyfrowanie danych uwierzytelniających dla wielu niezależnych dostawców chmurowych przy jednoczesnym zachowaniu możliwości natychmiastowego powrotu do bezpłatnego i bezpiecznego modelu lokalnego.

Niezwykle cennym elementem podnoszącym ogólne bezpieczeństwo procesu automatycznej refaktoryzacji byłoby zaimplementowanie podsystemu automatycznej weryfikacji regresji. Komponent ten po zatwierdzeniu zmian przez użytkownika w oknie różnicowym uruchamiałby w tle zestaw lokalnych testów jednostkowych. W przypadku wykrycia jakichkolwiek błędów lub niekompatybilności w strukturze

programu wtyczka mogłaby automatycznie wycofać naniesione poprawki lub przekazać logi kompilatora z powrotem do silnika sztucznej inteligencji w celu dokonania powtórnej, inteligentnej korekty kodu.

Ostatnim wartym uwagi kierunkiem rozwoju jest dostosowanie narzędzia do obsługi innych języków programowania. Architektura systemu oparta na niezależnych modułach analitycznych pozwala na relatywnie łatwe wdrożenie parserów dla ekosystemów takich jak `JavaScript` lub `Python`, w których problem dynamicznie dezaktualizujących się pakietów zewnętrznych jest równie powszechny i uciążliwy dla zespołów deweloperskich.

7.4. Zakończenie

W ramach pracy wykonano analizę cyklu życia współczesnych systemów informatycznych pod kątem mechanizmów powstawania długu technologicznego. W oparciu o wyniki analizy przedstawiono autorską koncepcję zarządzania tym zjawiskiem, która wprowadza usprawnienia w stosunku do istniejących narzędzi automatyzacji. Poprzez implementację prototypu rozwiązania podjęto próbę skrócenia czasu potrzebnego na identyfikację oraz eliminację przestarzałych zależności w projektach programistycznych.

W treści dokumentu szczegółowo omówiono mocne oraz słabe strony wypracowanej metody, a także wyzwania inżynierskie napotkane w trakcie pisania kodu. Wytworzona aplikacja ma charakter demonstracyjny i wymaga dalszych prac rozwojowych przed ewentualnym zastosowaniem w realnym środowisku komercyjnym.

Przyjęta struktura oparta na dwóch osobnych modułach potwierdziła swoją przydatność w praktyce. Komponent działający jako rozszerzenie dla środowiska IntelliJ IDEA pozwala na bezpieczne przekształcanie struktur językowych bezpośrednio w oknie edytora, podczas gdy program konsolowy koncentruje się na szybkim przetwarzaniu skompilowanych klas. Wydzielenie niezależnych komponentów zapewnia elastyczność całego rozwiązania i umożliwia jego rozszerzenie o kolejne funkcjonalności w przyszłości.

Zaproponowane podejście pozwala zespołom programistycznym na znacznie efektywniejsze kontrolowanie kondycji technicznej systemów oraz sprawniejsze reagowanie na wycofywanie wsparcia dla używanych bibliotek.

Bibliografia

- [1] Anthropic. The Claude Model Family. <https://www.anthropic.com/claude>. [Dostęp: 7 maja 2026].
- [2] Jinze Bai, Shuai Bai, Yunfei Chu, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023. <https://arxiv.org/abs/2309.16609> [Dostęp: 7 maja 2026].
- [3] Lennard Bonorden and Matthias Riebisch. In *API Deprecation: A Systematic Mapping Study*. International Conference on Software Maintenance and Evolution, 2022.
- [4] Gustavo Brito, Marco Tulio Valente, and Ricardo S. Terra. On the use of replacement messages in api deprecation: An empirical study. *Journal of Systems and Software*, 137:306–321, 2018.
- [5] Ward Cunningham. The wycash portfolio management system. *OOPSLA '92*, 4(2):29–30, 1992.
- [6] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall Professional, 2004. ISBN: 978-0131177055.
- [7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN: 978-0134757599.
- [8] Gemma Team, Google DeepMind. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024. <https://arxiv.org/abs/2403.08295> [Dostęp: 7 maja 2026].
- [9] Georgi Gerganov. llama.cpp: Port of Facebook’s LLaMA model in C/C++. <https://github.com/ggerganov/llama.cpp>. [Dostęp: 7 maja 2026].
- [10] GitHub. GitHub Copilot. <https://github.com/features/copilot>. [Dostęp: 7 maja 2026].
- [11] Google. Google Gemini. <https://gemini.google.com/>. [Dostęp: 7 maja 2026].
- [12] Hugging Face. Hugging Face – The AI community building the future. <https://huggingface.co/>. [Dostęp: 7 maja 2026].
- [13] JetBrains s.r.o. IntelliJ IDEA – the Leading Java and Kotlin IDE. <https://www.jetbrains.com/idea/>. [Dostęp: 7 maja 2026].
- [14] JetBrains s.r.o. IntelliJ Platform SDK Documentation. <https://plugins.jetbrains.com/docs/intellij/>. [Dostęp: 7 maja 2026].
- [15] JetBrains s.r.o. IntelliJ Platform SDK Documentation: Code Inspections. <https://plugins.jetbrains.com/docs/intellij/code-inspections.html>. [Dostęp: 7 maja 2026].
- [16] JRebel by Perforce. 2025 java developer productivity report, 2025. [Dostęp: 18 maja 2026].
- [17] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification*. Oracle America, Inc., 2023. <https://docs.oracle.com/javase/specs/jvms/se17/html/index.html> [Dostęp: 7 maja 2026].

- [18] LM Studio. LM Studio - Discover, download, and run local LLMs. <https://lmstudio.ai/>. [Dostęp: 7 maja 2026].
- [19] Stuart Marks. JEP 277: Enhanced Deprecation. <https://openjdk.org/jeps/277>, 2015. [Dostęp: 7 maja 2026].
- [20] Sofia Martinez, Luo Xu, Mariam Elnaggar, and Eman Abdullah Alomar. Software refactoring research with large language models: A systematic literature review. *The Journal of Systems and Software*, 235:112762, 2026.
- [21] Meta AI. Introducing Llama 3: The most capable openly available LLM to date. <https://llama.meta.com/>. [Dostęp: 7 maja 2026].
- [22] Moderne. OpenRewrite Documentation: Lossless Semantic Trees (LST). <https://docs.openrewrite.org/concepts-and-explanations/lossless-semantic-trees>. [Dostęp: 7 maja 2026].
- [23] MvnRepository. Spring Core Artifact - Maven Repository. <https://mvnrepository.com/artifact/org.springframework/spring-core>. [Dostęp: 7 maja 2026].
- [24] Ollama. Ollama - Get up and running with large language models locally. <https://ollama.com/>. [Dostęp: 7 maja 2026].
- [25] OpenAI. OpenAI Platform Documentation: Models. <https://platform.openai.com/docs/models>. [Dostęp: 7 maja 2026].
- [26] OpenRewrite. Upgrade to spring boot 3.0. https://docs.openrewrite.org/recipes/java/spring/boot3/upgradespringboot_3_0, 2026. Dostęp 28 maja 2026 r.
- [27] Oracle. javac - The Java Programming Language Compiler (Java SE 17). <https://docs.oracle.com/en/java/javase/17/docs/specs/man/javac.html>. [Dostęp: 7 maja 2026].
- [28] Oracle. jdepscan - Java Static Analysis Tool (Java SE 17). <https://docs.oracle.com/en/java/javase/17/docs/specs/man/jdepscan.html>. [Dostęp: 7 maja 2026].
- [29] David Lorge Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287. IEEE, 1994.
- [30] Tom Preston-Werner. Semantic Versioning 2.0.0. <https://semver.org/>. [Dostęp: 7 maja 2026].
- [31] Ameya Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2083–2107, 2018.
- [32] SonarSource. SonarQube - Clean Code & Code Quality Analysis. <https://www.sonarsource.com/products/sonarqube/>. [Dostęp: 7 maja 2026].
- [33] Sonatype. State of the Software Supply Chain. <https://www.sonatype.com/state-of-the-software-supply-chain/2024/scale>. [Dostęp: 7 maja 2026].
- [34] The Apache Software Foundation. Maven Central Repository Search. <https://central.sonatype.com/>. [Dostęp: 7 maja 2026].
- [35] Mariusz Trzaska. *Modelowanie i implementacja systemów informatycznych 2.0*. 2025. 487 s., ISBN: 978-83-976442-0-5.
- [36] Yisen Xu, Jinqiu Yang, and Tse-Hsun Chen. Swe-refactor: A repository-level benchmark for real-world llm-based code refactoring. *arXiv preprint arXiv:2602.03712*, 2026.

Spis rysunków

2.1	Zestawienie wersji biblioteki <i>spring-core</i> w repozytorium Maven Central, obrazujące problem luk w zabezpieczeniach w przestarzałych gałęziach. Źródło: [23].	9
2.2	Wycinek wygenerowanego grafu zależności w środowisku Intelij IDEA. Źródło: opracowanie własne.	9
2.3	Schemat powstawania konfliktu diamentowego w grafie zależności. Źródło: opracowanie własne.	10
2.4	Cykl życia interfejsu API w kontekście Semantycznego Wersjonowania. Źródło: opracowanie własne.	11
2.5	Widok reguły dotyczącej użycia przestarzałego kodu w interfejsie systemu SonarQube. Źródło: opracowanie własne.	12
2.6	Przykład wykrycia metody przestarzałej przez system SonarQube. Źródło: opracowanie własne.	12
2.7	Zestawienie przepisów dla framework'a Spring Boot dostępnych w OpenRewrite. Źródło: [26].	13
2.8	Aktywacja narzędzia OpenRewrite z poziomu IDE. Źródło: opracowanie własne.	13
2.9	Wizualizacja inspekcji statycznej w środowisku IntelliJ IDEA. Źródło: opracowanie własne.	14
3.1	Koncepcyjny schemat blokowy przepływu danych i logiki decyzji we wtyczce. Źródło: opracowanie własne.	17
4.1	Uproszczona struktura drzewa PSI dla wywołania metody. Źródło: opracowanie własne.	21
6.1	Główny panel wtyczki z widoczną tabelą zależności i przyciskami funkcyjnymi. Źródło: opracowanie własne.	47
6.2	Okno dialogowe umożliwiające wybór trybu skanowania projektu. Źródło: opracowanie własne.	48
6.3	Okno dialogowe przedstawiające raport złożoności aktualizacji zależności. Źródło: opracowanie własne.	48
6.4	Widok panelu wtyczki w zakładce Deprecated Methods dla <code>com.github.javaparser:3.26.3</code> . Źródło: opracowanie własne.	49
6.5	Szczegółowy widok informacji o przestarzałej metodzie po wybraniu elementu z tabeli dla metody <code>com.github.javaparser.ast.Node.getChildNodesByType</code> . Źródło: opracowanie własne.	49
6.6	Widok panelu wtyczki w zakładce Method Usages po użyciu opcji Fetch Details . Źródło: opracowanie własne.	50
6.7	Menu wyboru konkretnej strategii naprawczej w ramach modułu Build-in Refactor Selected . Źródło: opracowanie własne.	51
6.8	Zestawienie porównawcze struktury kodu źródłowego obejmujące stan pierwotny oraz trzy alternatywne rezultaty działania wbudowanego silnika regułowego. Źródło: opracowanie własne.	52
6.9	Interaktywne okno porównawcze prezentujące kod przed i po transformacji AI. Źródło: opracowanie własne.	52

Spis listingów

2.1	Konfiguracja narzędzia OpenRewrite w pliku <code>pom.xml</code> . Źródło: opracowanie własne. . .	13
2.2	Instalacja i wywołanie narzędzia OpenRewrite z poziomu interfejsu wiersza poleceń. Źródło: opracowanie własne	14
4.1	Przykładowy plik <code>plugin.xml</code> rejestrujący okno narzędziowe. Źródło: opracowanie własne.	20
4.2	Teoretyczny schemat ustrukturyzowanego zapytania API do modelu językowego wraz z hiperparametrami. Źródło: opracowanie własne.	23
5.1	Odczyt i parsowanie zależności z pliku <code>pom.xml</code> przy użyciu <code>MavenXpp3Reader</code> . Źródło: opracowanie własne.	26
5.2	Konstrukcja adresu URL do pobrania archiwum z kodem źródłowym. Źródło: opracowanie własne.	27
5.3	Dynamiczna konstrukcja adresu URL dla metadanych biblioteki. Źródło: opracowanie własne.	27
5.4	Przykładowa struktura odpowiedzi <code>maven-metadata.xml</code> pobranej z repozytorium. Źródło: opracowanie własne.	27
5.5	Strumieniowe przeszukiwanie katalogu oraz orkiestracja analizy pliku źródłowego. Źródło: opracowanie własne.	28
5.6	Algorytmy ekstrakcji uzasadnienia deprecjacji oraz rekomendowanych zamienników API. Źródło: opracowanie własne.	29
5.7	Implementacja wizytatora PSI wykrywającego przestarzałe metody i konstruktory. Źródło: opracowanie własne.	31
5.8	Proces bezpiecznej iteracji po plikach projektu z wykorzystaniem blokady odczytu. Źródło: opracowanie własne.	32
5.9	Konfiguracja strumienia klasy i wywołanie czytnika <code>ClassReader</code> . Źródło: opracowanie własne.	33
5.10	Implementacja klasy <code>ClassVisitor</code> śledzącej kontekst i delegującej zadania. Źródło: opracowanie własne.	33
5.11	Przechwytywanie instrukcji wywołań oraz śledzenie linii kodu w <code>MethodVisitor</code> . Źródło: opracowanie własne.	34
5.12	Hybrydowa weryfikacja statusu deprecjacji w klasie <code>BytecodeDeprecationScanner</code> . Źródło: opracowanie własne.	36
5.13	Implementacja metody <code>checkDeprecationInBytecode</code> oraz wizytatora <code>DeprecationCheckVisitor</code> . Źródło: opracowanie własne.	37
5.14	Konstrukcja zapytania do modelu AI z wykorzystaniem zebranego kontekstu. Źródło: opracowanie własne.	38
5.15	Przykład wygenerowanego zapytania (promptu) wysyłanego do LLM. Źródło: opracowanie własne.	40
5.16	Przykładowa odpowiedź wygenerowana przez model sztucznej inteligencji. Źródło: opracowanie własne.	40
5.17	Orkiestracja transformacji strukturalnych w klasie <code>RefactoringService</code> . Źródło: opracowanie własne.	41
5.18	Asynchroniczne pobieranie kontekstu i generowanie sugestii sztucznej inteligencji w klasie <code>AIRefactoringService</code> . Źródło: opracowanie własne.	42

5.19	Modyfikacja dokumentu tekstowego z wymuszeniem synchronizacji PSI w klasie <code>AI-RefactoringService</code> . Źródło: opracowanie własne.	44
5.20	Tworzenie panelu podglądu różnicowego. Źródło: opracowanie własne.	46
6.1	Ogólna składnia polecenia <code>scan</code> . Źródło: opracowanie własne.	53
6.2	Wywołanie polecenia <code>scan</code> dla zależności <code>javaparser-core</code> . Źródło: opracowanie własne.	53
6.3	Wynik polecenia <code>scan</code> prezentujący wycofane metody. Źródło: opracowanie własne.	53
6.4	Ogólna składnia polecenia <code>compare</code> . Źródło: opracowanie własne.	54
6.5	Wywołanie polecenia <code>compare</code> dla biblioteki <code>JUnit</code> . Źródło: opracowanie własne.	54
6.6	Wynik wywołanie polecenia <code>compare</code> dla biblioteki <code>JUnit</code> . Źródło: opracowanie własne.	54
6.7	Podstawowe wywołanie <code>scan-project</code> bez flag dodatkowych. Źródło: opracowanie własne.	56
6.8	Wynik tekstowy polecenia <code>scan-project</code> w trybie szybkim. Źródło: opracowanie własne.	56
6.9	Wywołanie pełnego skanowania projektu z flagą <code>--full</code> . Źródło: opracowanie własne.	57
6.10	Dodatkowy element raportu otrzymanego z wykorzystaniem polecenia <code>scan-project --full</code> . Źródło: opracowanie własne.	57
6.11	Wywołanie skanowania z formatowaniem JSON oraz pełnym opisem. Źródło: opracowanie własne.	57
6.12	Rezultat skanowania projektu z użyciem flag <code>--json</code> oraz <code>--full</code> . Źródło: opracowanie własne.	58
6.13	Inicjalizacja modułu konsolowego w trybie interaktywnym. Źródło: opracowanie własne.	58
6.14	Rezultat uruchomienia modułu konsolowego w trybie interaktywnym. Źródło: opracowanie własne.	59

Spis tabel

5.1	Składniki kontekstu wstrzykiwane do zapytania dla modelu AI. Źródło: opracowanie własne.	39
5.2	Zestawienie komponentów interfejsu graficznego wykorzystanych we wtyczce. Źródło: opracowanie własne.	45