POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Maciej Kanecki
Nr albumu s15602

System wspierający integracje pomiędzy narzędziami developerskimi

Praca Magisterska

Dr inż. Mariusz Trzaska

Warszawa, lipiec 2024

Streszczenie

Tematem pracy jest opracowanie generycznego sposobu na integracje ze sobą najczęściej wykorzystywanych przez developerów narzędzi, takich jak Jira czy Git. Znamienita większość projektów związanych z wytwarzaniem oprogramowania jest realizowana w obecnych czasach w zespołach. Oznacza to, że w celu usprawnienia współpracy i zminimalizowania ryzyka błędów konieczna jest implementacja procesów pozwalających w sposób jawny określić kto jest za dany fragment systemu odpowiedzialny. Tego typu procedury pozwalają potem firmom określić, na jakim etapie pracy jest dany programista i umożliwiają połączenie wytworzonego przez niego fragmentu z elementami dostarczonymi przez innych członków zespołu. Często w celu organizacji tego typu pracy wykorzystywane są systemy Jira i Git (Zazwyczaj zaimplementowany w formie internetowej platformy jak GitHub czy Gitlab). Niestety ze względu na ich niezależność często wymagana jest spora ilość ręcznych zmian ze strony developerów w celu utrzymania systemów w stanie zsynchronizowanym. Zaproponowane w tej pracy narzędzie pozwala uniknąć tego problemu i automatycznie przełożyć wydarzenia w jednym systemie na akcje podjęte w drugim. Pozwala to na usprawnienie pracy programistów i oszczędza firmie część ich czasu.

Słowa kluczowe: Jira, Git, zarządzanie procesami, integracja

Spis treści

1. Wstęp

Wzrost poziomu skomplikowania projektów w sektorze IT na przestrzeni ostatnich kilkunastu lat wywołał zapotrzebowanie na narzędzia wspierające koordynacje większych zespołów i pozwalający na tworzenie dużo bardziej złożonych systemów informatycznych. To wtedy powstały właśnie Jira [16], Git [17] i dalsze jego implementacje w formie platform takich jak GitHub [18] czy Bitbucket [19], które na tym etapie są nieodzownym elementem większości nowoczesnych projektów informatycznych.

Rezultatem coraz większego polegania przez programistów na różnego rodzaju systemach wspierających ich pracę, jest wzrost ilości zależności między tymi właśnie programami. Może to wymuszać na pracownikach manualne interwencje w celu utrzymania spójności między narzędziami. Z perspektywy zarządzania projektami, optymalne pod względem utylizacji zasobów ludzkich byłoby ograniczenie bezproduktywnej pracy programistów do minimum, co jest celem opisywanym w tej pracy.

W celu zaadresowania opisanego powyżej problemu, oprogramowanie pozwoli nam przyjąć zapytania POST wygenerowane przez różnego rodzaju wydarzenia w innych systemach wykorzystywanych w projekcie. Zostaną one następnie przetworzone w oparciu o konfiguracje dostarczoną przez użytkownika a następnie przekazanie do docelowego systemu. Pozwoli nam to zautomatyzować powtarzalne czynności i zminimalizować ilość czasu traconą przez developerów.

1.1. Cel pracy

Celem pracy jest zredukowanie bezproduktywnego, a koniecznego, wysiłku wymaganego ze strony developerów w celu utrzymania pożądanego stanu narzędzi kontroli procesu. Pozwala to programistom skupić się na zadaniach zleconych przez klienta tym samym potencjalnie redukując ogólne koszty projektu poprzez oszczędność czasu. Zaproponowane rozwiązanie jest generyczne i pozwala na konfiguracje akcji wykonywanych przez program przy pomocy plików z instrukcjami.

Ze względu na zastosowane podejście do definicji przepływów w programie, możliwe też jest projektowanie przepływów przez użytkowników nie posiadających znacznej wiedzy programistycznej.

1.2. Rozwiązanie przyjęte w pracy

Do definiowania procesów w programie wykorzystywane są specjalnie w tym celu utworzone pliki YAML. Oprogramowanie jest napisane w Java w technologii Spring [12] z wykorzystaniem Apache Camel [1] do przetwarzania zapytań a w celu przygotowania dokumentacji utworzonych klas i procesów developerskich wykorzystana jest notacja UML.

1.3. Rezultaty pracy

Wynikiem pracy jest opracowanie sposobu na generyczną integracje wielu narzędzi na raz. Opisany w niej prototyp ma za zadnie przedstawić potencjalne rozwiązanie tego problemu i wynikających z tego tytułu dla firmy potencjalnych zysków. Pozwala to w znaczny sposób ograniczyć czas tracony przez programistów na wykonywanie bezproduktywnych akcji w celu utrzymania spójności między narzędziami dostępnymi w projekcie podnosząc jego wydajność.

1.4. Organizacja pracy

W sekcji 2 pracy został przedstawiony opis problemów z jakimi mierzyć się muszą developerzy w trakcie codziennej pracy na projektach informatycznych oraz krótkie podsumowanie innych badań potwierdzających tezę autora.

W sekcji 3 przedstawione zostały inne dostępne już na rynku rozwiązania oraz przeprowadzona została analiza zalet i wad każdego z nich. Rezultatem tej analizy są kluczowe założenia jakie opisywane w tej pracy rozwiązanie musi spełnić, żeby być w stanie odnaleźć sukces na rynku.

Sekcja 4 zawiera opis poszczególnych technologii wykorzystanych do przygotowania pracy, wraz z opisem procesu decyzyjnego autora, którego rezultatem był wybór przedstawionych rozwiązań.

W sekcji 5 dokumentu znajduje się szczegółowy opis poszczególnych elementów przygotowanych w trakcie tworzenia prototypu mającego na celu zaimplementować zdefiniowane w sekcjach 2 i 3 założenia. Przedstawiane komponenty systemu zostały udokumentowane przy pomocy diagramów UML a ich kluczowe metody zostały szczegółowo opisane w wyszczególnionych sekcjach. W ramach przygotowanego prototypu dostarczone zostały następujące komponenty:

- Request Adapter SPI,
- Request Adapter Core,
- Przykładowa *wtyczka* wejścia,
- Przykładowe *rozszerzenie* wyjścia.

W rozdziale 6 przedstawione zostały informacje o konfiguracji rozwiązania i tym jak jego implementacja może uprościć przykładowy proces biznesowy przedstawiony w tej sekcji.

Na końcu znajdziemy podsumowanie wyników powstałego prototypu wraz z proponowanymi następnymi krokami które mogłyby pozwolić na dalszy rozwój zaprezentowanego rozwiązania.

2. Opis problemu

Branża IT jest jednym z najszybciej rozwijających się sektorów gospodarczych w dwudziestym pierwszym wieku, z ogromną przestrzenią na innowacyjne projekty, ale i jednocześnie ogromną rywalizacją pomiędzy firmami zajmującymi się pracą programistyczną. W celu podniesienia konkurencyjności wśród firm działających na rynku kluczowe wydaje się dążenie do maksymalizacji ilości czasu jaką programiści mogą spędzić na faktycznym programowaniu. Ze względu na wzrost poziomu skomplikowania projektów w branży konieczne było też wprowadzenie dodatkowych systemów kontroli. Tego typu oprogramowanie może przełożyć się na dodatkową prace dla programistów, tym samym redukując ich produktywność. Jednocześnie może ono też podnieść szanse na powodzenie projektu ze względu na większa ilość kontroli nad procesem wytwarzania oprogramowania.

W 2019 roku, firmy The New Stack i Tidelift w ramach artykułu „How Much Time Do Developers Spend Actually Writing Code?" [1] przeprowadziły ankietę mającą na celu weryfikacje, ile czasu developerzy faktycznie spędzają na pisaniu kodu. Z odpowiedzi ich prawie 400 respondentów wynikało, że średnio developerzy spędzają około 32% swojego czasu na programowaniu a aż 23% na spotkaniach czy 19% na zarządzaniu kodem. W oparciu o powyższe statystyki można wyciągnąć wniosek, że na rynku bardzo duży potencjał uzyskać mogą firmy, które znajdą sposób na zmniejszenie ilości czasu jaki developerzy poświęcać muszą na zadania nie związane z programowaniem.

Autorzy badania "Today was a Good Day: The Daily Life of Software Developers" [2] doszli w tym samym roku do podobnych wniosków. Swoje dane zebrali oni od grupy 5971 programistów firmy Microsoft, którzy przedstawili, jak wygląda ich standardowy, dobry i zły dzień pracy. Według ich respondentów, w dobry dzień spędzają oni maksymalnie 50% swojego czasu na zadaniach technicznych z czego tylko 32% na faktycznym programowaniu czy rozwiązywaniu błędów w oprogramowaniu. Odkryli też, że strata czasu wywołana oderwaniem programisty od pracy nie jest ograniczona tylko do samego źródła przerwania pracy np.: spotkania czy pytań ze strony innych programistów, ale wykracza poza ten okres czasu. Według ich badania, średnio, w dobry dzień, programiści potrzebują około 8 minut, żeby odzyskać pełną produktywność w wykonywanej przez nich pracy po wystąpieniu wydarzenia przerywającego ich skupienie. W oparciu o powyższe wyniki, rozwiązania redukujące zbędne odciąganie pracowników od wykonywanych zadań i pozwalające im możliwie największą ilość czasu poświęcić na pracę ściśle developerską mają duży potencjał podnieść konkurencyjność firm je implementujących.

Aby uniknąć przedstawionych wyżej problemów przygotowane został opisane w tej pracy rozwiązanie. Ma ono na celu umożliwić zautomatyzowanie czynności powtarzalnych takich jak przestawianie statusów na Jirze w momencie, kiedy developer rozpocznie prace nad daną funkcjonalnością, czy zmianę osoby przypisanej do danego zadania w momencie, kiedy w systemie kontroli kodu zostanie ono przekazane do recenzji. Specjalna uwaga została też poświęcona zapewnieniu generycznej natury zaprojektowanego rozwiązania czy możliwości rozszerzenia go o nowe systemy poprzez zastosowaną architekturę opartą o wtyczki.

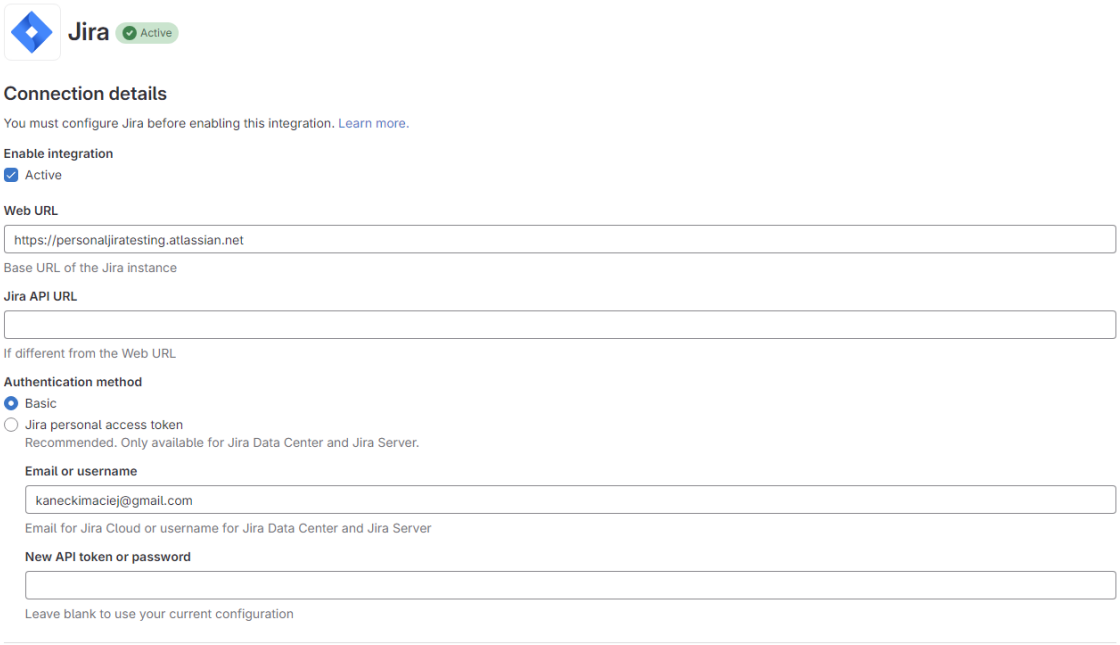
3. Istniejące rozwiązania

Celem tego rozdziału jest prezentacja dostępnych obecnie na rynku rozwiązań konkurencyjnych, analizy ich zalet oraz wad i porównanie do zaproponowanego prototypu. W poniższych podrozdziałach przedstawione zostały wybrane projekty.

3.1. Wtyczka do GitLab „Jira issues integration”

Jednym z obecnych dostępnie na rynku rozwiązań jest *plugin* dostępny z poziomu platformy GitLab, pozwalający na podstawową integrację z wybraną instancją Jira. Po skonfigurowaniu wtyczki dla wykorzystywanej instancji (przedstawionej na Rysunek 1), poprzez podanie danych jak Web URL oraz danych niezbędnych do autentykacji użytkownika uzyskujemy dostęp do wszystkich funkcjonalności opisywanego rozwiązania. Główne z nich to między innymi:

- automatyczne zamykanie *ticketu*, w momencie zakończenia przez programistów walidacji kodu w oparciu o zdefiniowane statusy,
- automatyczne podłączanie linków do *merge request* lub *commit* z GitLab na Jira.



Rysunek 1. Panel konfiguracyjny pluginu wtyczki Jira issues integration

Zalety:

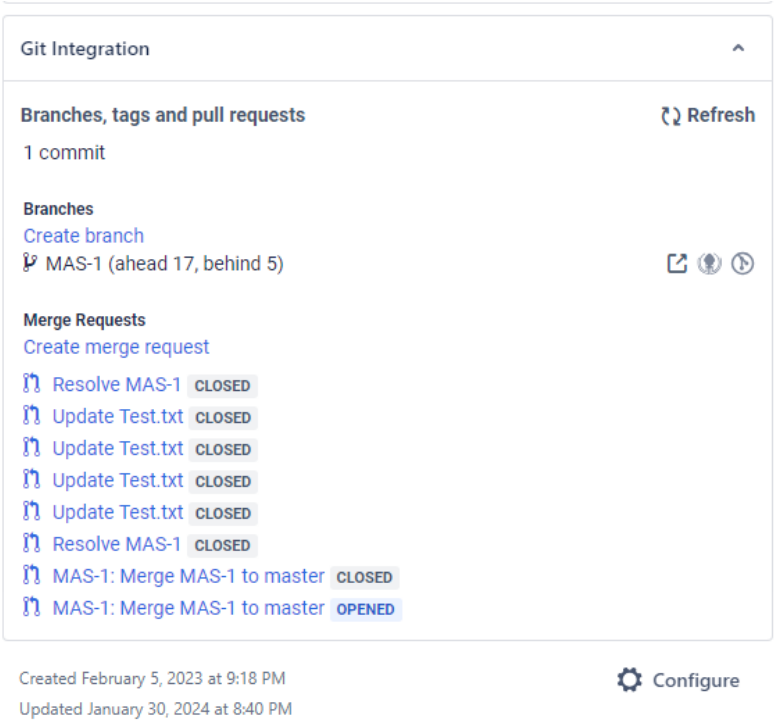
- Łatwość instalacji oraz konfiguracji rozwiązania pozwala na szybką implementację rozwiązania w projekcie,
- Dostępne akcje są w znacznej mierze skonfigurowane automatycznie, niektóre wymagają minimalnej konfiguracji przez GUI GitLab.

Wady:

- Rozwiązanie dedykowane dla platformy GitLab, nie jest wieloplatformowe i nie może być wykorzystane dla innych systemów,
- Pozwala na podłączenie tylko do jednej instancji Jira uniemożliwiając tym samym jednoczesne wprowadzanie modyfikacji na wielu platformach na raz,
- Brak możliwości konfiguracji dodatkowych akcji, dostarczony zestaw funkcji nie może zostać rozszerzony co może być ograniczające dla niektórych użytkowników.

3.2. Aplikacja do Jira „Git Integrations for Jira”

Komercyjny projekt pozwalający na poszerzenie ilości informacji dostępnych o akcjach podejmowanych na platformie Git z poziomu Jira (przykład przedstawiony na Rysunek 2). Udostępnia informacje na temat *branchy*, *merge request-ów* oraz *commit-ów* powstających na git dla danego *ticketu.* Poszerza tym samym zakres informacji jakie menadżer projektu posiada o stanie pracy swoich współpracowników. Pozwala też na zdalne zakładanie gałęzi czy *pull request*, potencjalnie ułatwiając zarządzanie nazewnictwem i *branch-ami* jakie powstają w projekcie. Nie posiada niestety wbudowanych funkcji pozwalających na automatyzacje procesów, zamiast tego polegając na dostępnej w Jirze automatyzacji i narzędziach developerskich, które ograniczone są do bardzo małego zakresu systemów.



Rysunek 2. Przykładowy panel Git Integration dla zadania

Zalety:

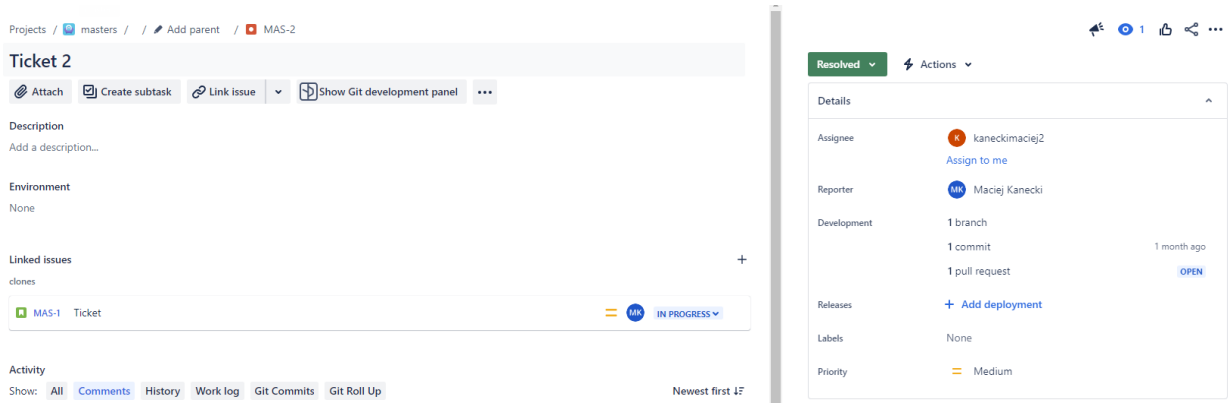
- Bardzo prosta instalacja i integracja z istniejącym projektem Jira,
- Możliwość interakcji z narzędziami Git przez osoby bez wiedzy technicznej przy pomocy GUI,
- Wysokiej jakości dokumentacja i wsparcie techniczne, ułatwiające konfiguracje rozwiązania do własnych potrzeb.

Wady:

- Narzędzie komercyjne, z kosztem subskrypcji rosnącym wraz ze zwiększającą się ilością użytkowników, co może być nie do zaakceptowania przez mniejsze i średnie przedsiębiorstwa,
- Brak możliwości wykonywania akcji bezpośrednio na Jira w oparciu wydarzenia na GIT,
- Aplikacja wbudowana w ekosystem firmy Atlassian, nieoferująca wsparcia dla innych systemów wyjściowych.

3.3. Aplikacja do Jira „Github for Jira”

Aplikacja dostępna z poziomu Jira, pozwalająca na ściślejszą integrację z projektem realizowanym na platformie Github. Pozwala poszerzyć ilość informacji dostępnych w ramach *ticketów* znajdujących się w systemie poprzez dodanie sekcji Development przedstawionej na Rysunek 3, przedstawiającej informacje na temat *pull request-ów, commit-ów* oraz gałęzi, które obecnie powiązane są z opracowywanym zadaniem. Dzięki dostarczonemu przez wtyczkę panelowi, użytkownik uzyskuje szybki dostęp do akcji możliwych do wykonania na GitHub, takich jak otwieranie czy dodawanie nowych *merge request-ów* czy otwieranie istniejących branchy.



Rysunek 3. Przykładowy ticket z panelem Development

Zalety:

- Prosta instalacja poprzez udostępniane przez Jira GUI do instalacji nowych aplikacji,
- Aplikacja rozszerza funkcjonalność zadań widocznych z poziomu Jira, jednocześnie będąc darmowym rozwiązaniem. Jeżeli pozwoli na to polityka organizacyjna może uprościć część akcji w projekcie jednocześnie nie podnosząc kosztów.

Wady:

- Brak możliwości wykonywania akcji w Jira na podstawie wydarzeń w Git, narzędzie jest raczej skupione na udostępnianiu nowych funkcji użytkownikowi w celu osobistej realizacji zadań,
- Wtyczka skupiona tylko na jednej platformie, w przypadku wyboru innych narzędzi Git konieczne byłoby wykorzystywanie i konfigurowanie kolejnych wtyczek,
- Narzędzie ściśle zintegrowane z ekosystemem Atlassian, nie jest możliwe wykorzystanie go na innych platformach o podobnym przeznaczeniu.

3.4. Podsumowanie

Dostępne obecnie na rynku rozwiązania nie są generyczne a w swojej implementacji skupiają się na konkretnych typach systemów. Dostarczane w nich GUI zdecydowanie ułatwia obsługę, ale niestety oferowane przez nie funkcjonalność jest bardzo podstawowa i może nie być zadowalająca dla bardziej wymagających zespołów. W oparciu o przykłady przedstawione w sekcjach 3.1, 3.2 i 3.3 dojść można do wniosku, że potencjalnie popularność mogłoby zdobyć rozwiązanie oferujące możliwość integracji z większym zakresem systemów oraz szerszą gamą akcji dostępnych do wykonania.

4. Wykorzystane technologie

W tej sekcji opisane zostały kluczowe technologie wykorzystane w trakcie przygotowywania pracy, poczynając od wybranego języka programowania a kończąc na popularnych na rynku szablonach do wytwarzania oprogramowania.

4.1. Java

W celu osiągnięcia przedstawionych w pracy celów konieczne jest wybranie języka programowania dobrze znanego przez obecnie pracujących na rynku developerów. Tylko w takiej sytuacji będziemy w stanie utworzyć społeczność developerską pozwalającą na dalszy rozwój przedstawionego rozwiązania, chociażby poprzez wytwarzanie nowych wtyczek do obsługi większej gamy systemów.

Aby umożliwić wykorzystywanie oprogramowania w jak największej ilości miejsc pracy, konieczne jest też, żeby przygotowany prototyp, jak i wykorzystywane przez niego wtyczki, były wieloplatformowe. Pozwoli to jeszcze bardziej obniżyć bariery mogące przeszkodzić wybranym w firmom w adopcji przedstawionego rozwiązania.

Ze względu na pozostałe w tej sekcji wymienione powody, jak i znajomość języka przez autora, wybrany do implementacji prototypu został język Java w wersji SE 11 (LTS). Jest on jednym ze standardów dostępnych obecnie na rynku i został wykorzystany w wielu komercyjnych rozwiązaniach na przestrzeni ostatnich lat. Wersja 11 jest też obecnie najczęściej wykorzystywaną przy obecnych implementacjach według raportu opublikowanego przez firmę *new relic* [3] w 2023 roku, z których wynika, że aż 56% rozwiązań produkcyjnych jest obecnie realizowanych na tej platformie.

4.2. IntelliJ IDEA

Do realizacji prototypu opisywanego w pracy wybrane zostało IntelliJ IDEA ze względu na jego znajomość przez autora pracy. Od wielu lat cieszy się rosnącą popularnością [4] ze względu na łatwość w obsłudze i przejrzystość interfejsu. W związku z jego powszechnym stosowaniem na rynku, dostępne jest też dużo materiałów wspierających pracę z nim i pozwalających na szybsze rozwiązanie potencjalnych problemów.

4.3. Lombok

W celu uproszczenia struktury pisanego kodu i uniknięcia jak największej ilości *boilerplate* kodu wykorzystany w projekcie zostało rozszerzenie Lombok [5]. Pozwala on nam uniknąć ręcznego pisania nadmiernej ilości metod *get* i *set,* które są standardową praktyką w języku Java, a zamiast tego automatycznie generuje je w oparciu o strukturę przygotowanych klas. Oczekiwane przez nas zachowanie wtyczki zdefiniować możemy poprzez *tagi* zamieszczane bezpośrednio nad definicją przygotowywanej przez nas klasy co zostało przedstawione na Listing 1.

Listing 1. Przykład tagów Lombok

```
...
@Data
@NoArgsConstructor
@AllArgsConstructor
public abstract class RepositoryInputEvent implements InputEvent {
    public enum EventType { PUSH, MERGE}
    public enum MergeRequestStatus {CLOSED, CAN_BE_MERGED,
CANNOT_BE_MERGED, MERGED, VALIDATION}
    private MergeRequestStatus mergeRequestStatus;
    private EventType eventType;
    private String repositoryProjectId;
    private String repositoryProjectName;
    private URL repositoryURL;
    private String repositoryDescription;
    private String repositoryDefaultBranch;
    private String userId;
    private String userUserName;
    private URL mergeRequestUrl;
    private String sourceBranch;
    private String targetBranch;
    private String mergeTitle;
    private boolean workInProgress;
    private String assigneeId;
    private String assigneeName;
    private String authorId;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
}
```

W opisie działania *pluginu* skupimy się na adnotacji @Data [6]. W przygotowanym prototypie jego głównym celem było wygenerowanie wcześniej wspomnianych metod *get* i *set*. Na Listing 2 przedstawione zostały wybrane wygenerowane w ten sposób funkcje w celu zobrazowania sposobu działania wtyczki.

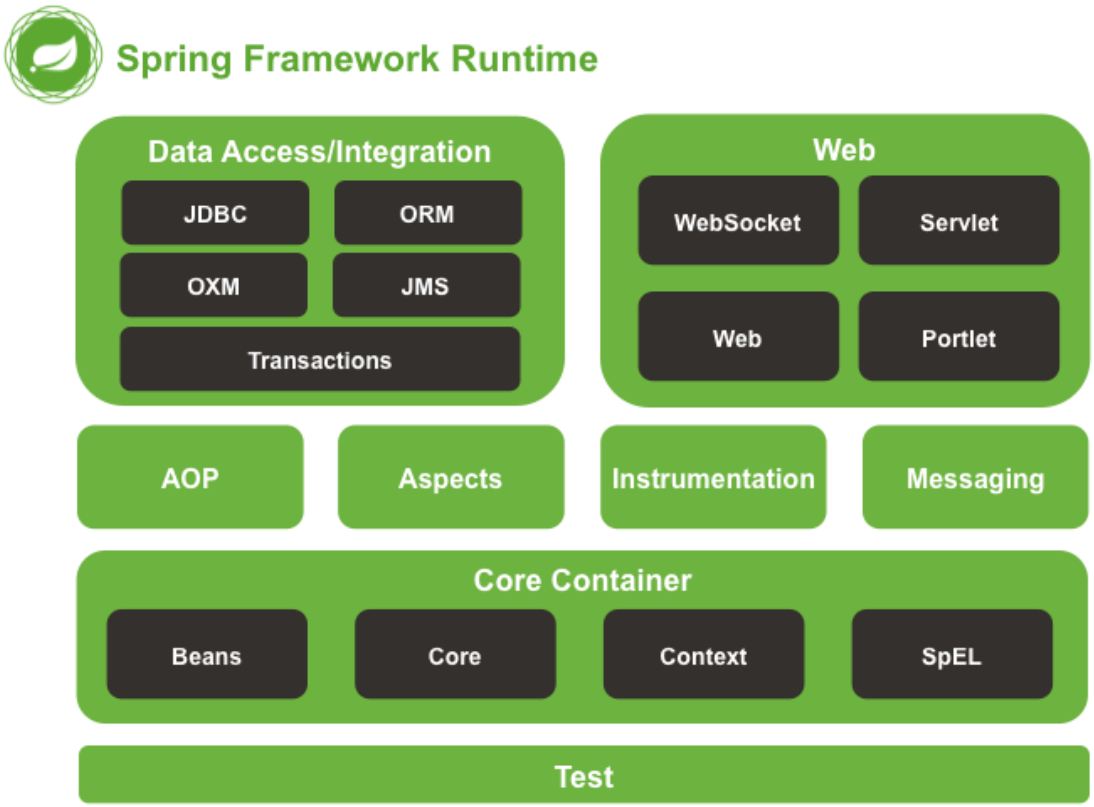
Listing 2. Zdekompilowana klasa RepositoryInputEvent

```
...
public MergeRequestStatus getMergeRequestStatus() {
    return this.mergeRequestStatus;
}

public EventType getEventType() {
    return this.eventType;
}

public String getRepositoryProjectId() {
    return this.repositoryProjectId;
}

public String getRepositoryProjectName() {
    return this.repositoryProjectName;
}


...

public void setMergeRequestStatus(MergeRequestStatus
mergeRequestStatus) {
    this.mergeRequestStatus = mergeRequestStatus;
}

public void setEventType(EventType eventType) {
    this.eventType = eventType;
}

public void setRepositoryProjectId(String
repositoryProjectId) {
    this.repositoryProjectId = repositoryProjectId;
}

public void setRepositoryProjectName(String
repositoryProjectName) {
    this.repositoryProjectName =
repositoryProjectName;
}
...
```

4.4. Spring Boot

Jeden z cieszących się największą popularnością wśród użytkowników *frameworków* języka Java jest zdecydowanie Spring. Służy on do wytwarzania biznesowych aplikacji webowych i wspiera w tym programistów poprzez dostarczenie w lekkim pakiecie całego systemu obsługi infrastruktury aplikacji, pozwalając w ten sposób programistom skupić się na wytwarzaniu nowych funkcji oprogramowania. Dzięki wykorzystaniu POJO (*Plain Old Java Object*) do konfiguracji działania systemu, Spring zapewnia nam elastyczność poprzez brak ścisłego powiązania pomiędzy wykorzystywanym *frameworkiem* a logiką biznesową. Na Rysunek 4 przedstawiony został podział modułów dostępnych w

Spring. Na potrzeby przygotowanego prototypu zostały wykorzystane głównie moduły *Beans* i *Core* do obsługi fundamentalnych części *frameworka* i wstrzykiwania zależności.



Rysunek 4. Diagram modułów spring. Źródło: [20]

W ramach prototypu wykorzystany został wariant *frameworka* Spring w formie SpringBoot, pozwalającej na szybką konfigurację podstawowych funkcjonalności aplikacji poprzez GUI zamieszczony na stronie Spring Initializr [7], dostarczający potem gotową do pobrania paczkę z podstawową konfiguracją aplikacji.

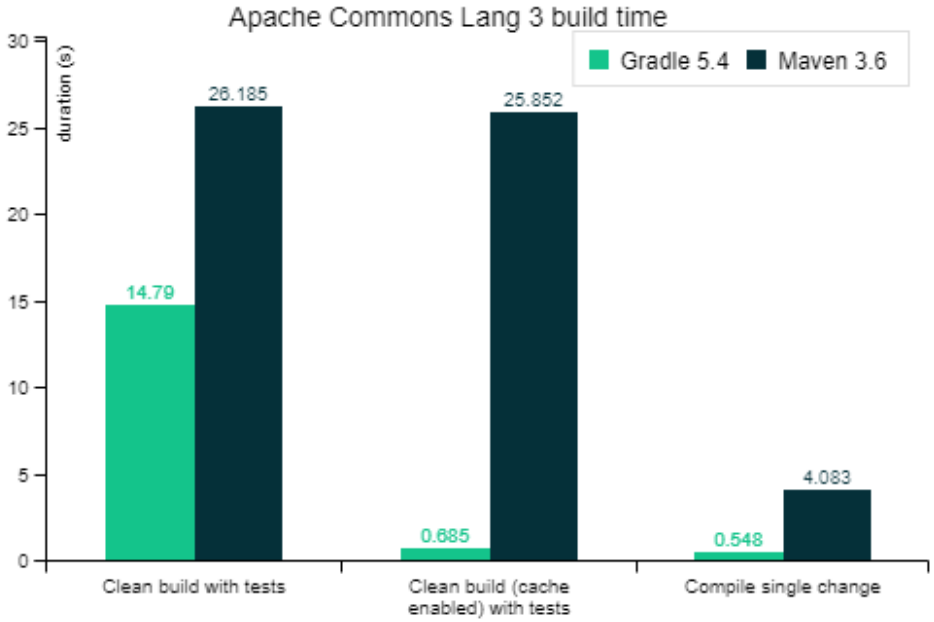
4.5. Gradle

Gradle [8] jest jednym z najpopularniejszych narzędzi do zarządzania zależnościami oraz automatyzacji budowy i instalacji paczek w języku Java. Posiada ono kilka kluczowych przewag nad konkurencyjnymi rozwiązaniami takimi jak Maven czy Ant i jest swego rodzaju hybrydą podejść reprezentowanych przez oba te narzędzia. Na Tabela 1 przedstawione zostały wybrane różnice pomiędzy Gradle a Maven które zadecydowały za wyborem tego właśnie narzędzia na potrzeby opisywanego w tej pracy prototypu.

Tabela 1. Porównanie Gradle i Maven

	Gradle	Maven
Sposób definiowania zadań	*Domain Specific Language* oparty o Groovy	XML
Sposób egzekucji zadań	Graf zadań	liniowy
Kompilacja przyrostowa	Tak	Nie

Konfiguracja oparta o DSL (*Domain Specific Language)* defnicje napisane w gradle są dużo bardziej zwięzłe i czytelniejsze, oszczędzając tym samym czas w trakcie pracy nad projektem. Oferowana przez Gradle kompilacja przyrostowa, osiągnięta poprzez weryfikacje konieczności ponownego wykonania poszczególnych zadań, również pozwala znacząco skrócić czas oczekiwania na otrzymanie budowanej przez nas paczki, w stosunku do Maven. Na Rysunek 5 przedstawione zostało porównanie wydajności obu tych rozwiązań, w oparciu o testy wykonane przez twórców oficjalnej dokumentacji Gradle [9], pozwalające dokładniej zobrazować przewagę tego narzędzia.



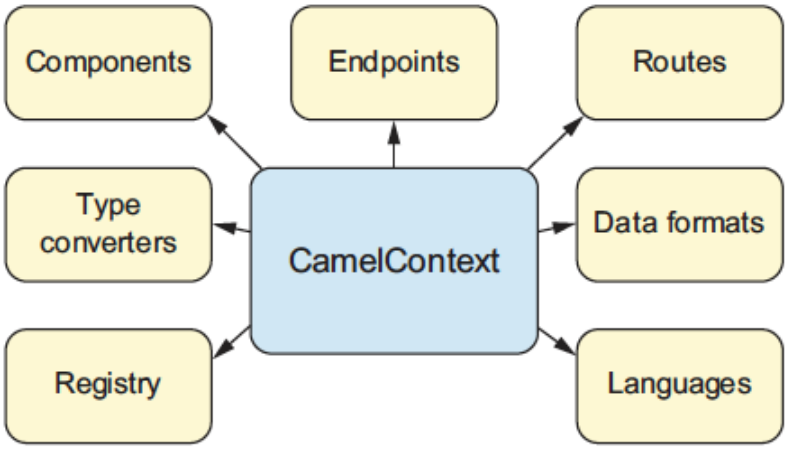
Rysunek 5. Porównanie wydajności Gradle i Maven. Źródło: [9]

4.6. Apache Camel

Kluczowy z perspektywy dostarczanego prototypu *open source* pakiet od grupy Apache, którego głównym zadaniem jest wspieranie programistów w implementacji *Enterpirse Integration Patterns* (EIP) i wykorzystywaniu tych schematów w obsłudze wszystkich warstw transportu. W kolejnych paragrafach opisane zostały najważniejsze dla działania prototypu elementy modułu Camel Core wraz z opisem sposobu ich wykorzystania w programie.

4.6.1 CamelContext

Jest to system wykonawczy modułu Camel Core, przechowujący wszystkie informacje na temat wykorzystywanej konfiguracji. Na Rysunek 6 przedstawiona została lista podstawowych serwisów przechowywanych przez *CamelContext* a w Tabela 2 ich przeznaczenie.



Rysunek 6. Struktura CamelContext. Źródło: [21]

Tabela 2. Przeznaczenie serwisów w Apache Camel

Serwis	Przeznaczenie
Registry	API do uzyskiwania informacji o *Bean'ach* zarejestrowanych w wykorzystywanym *frameworku*.
Type converters	Odpowiada za konwersje typów pomiędzy formatem źródłowym zapytania a docelowym wskazanym przez programistę.
Components	Pozwalają na podłączenie systemu do specyficznych technologii wejściowych/wyjściowych. Podstawowe komponenty są wbudowane w system i oferują możliwość integracji z opisywanymi przez nie protokołami przy wykorzystaniu domyślnej konfiguracji.
Endpoints	Zdefiniowane w obrębie komponentów ścieżki umożliwiające systemom zewnętrznym wysłanie wiadomości do aplikacji implementującej Camel.
Routes	Opisuje ścieżkę jaką musi przejść wiadomość dostarczona do systemu aż do miejsca docelowego.
Data formats	Odpowiada za *Marshalling/Unmarshalling* ciągu bitów na jeden z predefiniowanych w Camel typów.
Languages	Przechowuje informacje o załadowanych językach skryptowych wykorzystywanych do definicji do definicji przepływów w Camel DSL

W opisywanym prototypie i dostarczanych wtyczkach Apache Camel wykorzystywany jest do *routingu* przychodzących zapytań z systemów źródłowych oraz ich przekształcaniu na postać akceptowalną przez systemy wyjściowe.

4.6.2 Camel Routes

Camel *route* opisuje w jaki sposób przyjęte mają zostać przychodzące do systemu zapytania poprzez zdefiniowanie serii kroków które mają być podjęte dla każdego z nich. W celu ich implementacji rozszerzyć możemy klasę *RouteBuilder* lub przygotować odpowiednio sformatowany plik konfiguracyjny w formacie XML czy YAML. Na Listing 3 przedstawiona została najbardziej podstawowa postać *route* obsługująca komunikacje w obrębie tego samego *CamelContext*

Listing 3. Przykład Camel Route

```
from( uri: "direct:source")
        .to( uri: "direct:destination");
```

4.6.3 Camel Processor

W obrębie zdefiniowanych *Camel Routes* możemy następnie zmodyfikować postać przychodzącego zapytania w dowolny sposób. Efekt ten uzyskiwany jest poprzez dołączanie do przepływu zapytania specjalnie w tym celu napisanych procesorów, definiujących logikę jaka ma być zaaplikowana dla każdego z przychodzących zestawów danych. Procesory zdefiniować możemy poprzez specjalnie w tym celu przygotowany kod w języku Java, implementujący interfejs *Processor* lub, tak jak w przypadku *Camel Routes*, w obrębie pliku konfiguracyjnego XML lub YAML przy użyciu odpowiedniej składni. Na Listing 4 przedstawiona została przykładowa implementacja *Camel Processor* przy pomocy wyrażenia lambda. W przypadku przedstawionym na przykładzie, procesor odpowiedzialny jest za utworzenie nowego nagłówka które będzie pomocny na dalszych etapach procesowania zapytania.

Listing 4. Przykład wykorzystania procesora do modyfikacji zapytania

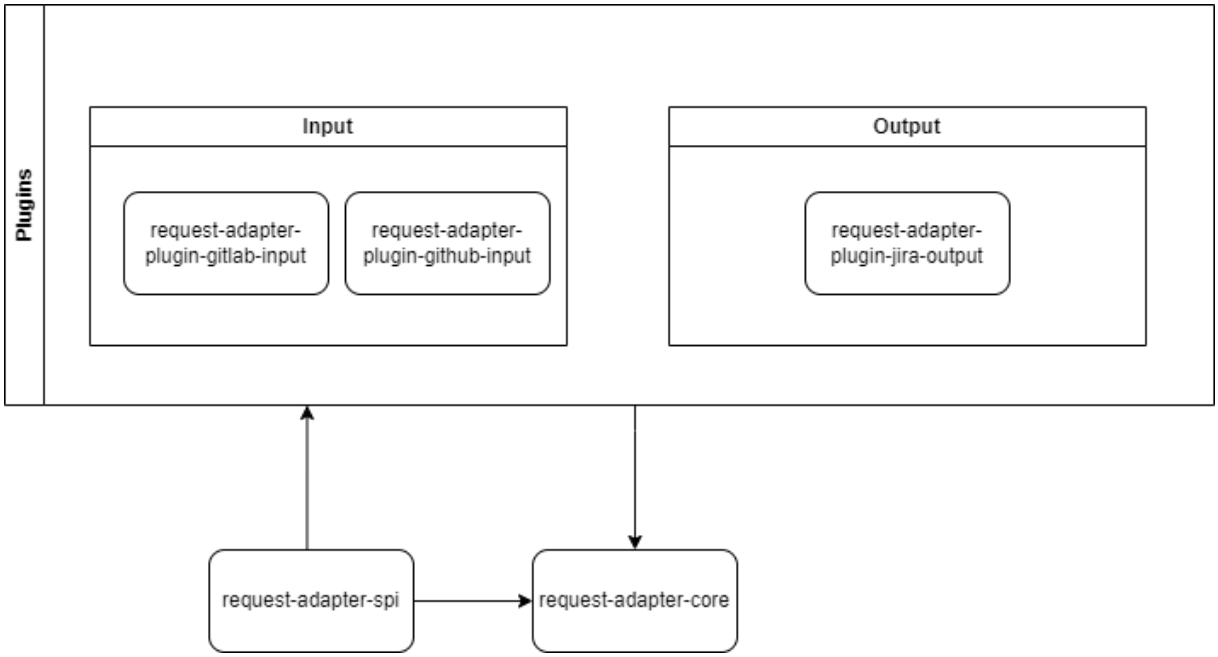
```
from( uri: "direct:input")
        .process(exchange -> {
            String requestSource = this.getRequestSource(exchange);
            exchange.getIn().setHeader(
                    INPUT_TYPE,
                    this.pluginLoader.getInputProcessorFactoryMap() Map<String, InputProcessorFactory>
                            .get(requestSource) InputProcessorFactory
                            .build() InputProcessor<InputEvent>
                            .getInputType()
            );
        }).toD( uri: "direct:${header." + INPUT_TYPE + "}");
```

4.6.4 Camel Exchange

Uniwersalny interfejs potrafiący przyjąć dane pozwalająca na reprezentacje przychodzącego zapytania w systemie i, jeśli takowa jest dostępna, odpowiedzi na nie. To właśnie obiekty implementujące interfejs exchange są modyfikowane w ramach prototypu w celu transformacji zapytania przychodzącego na format akceptowalny przez systemy docelowe.

5. Opis architektury przygotowanego prototypu

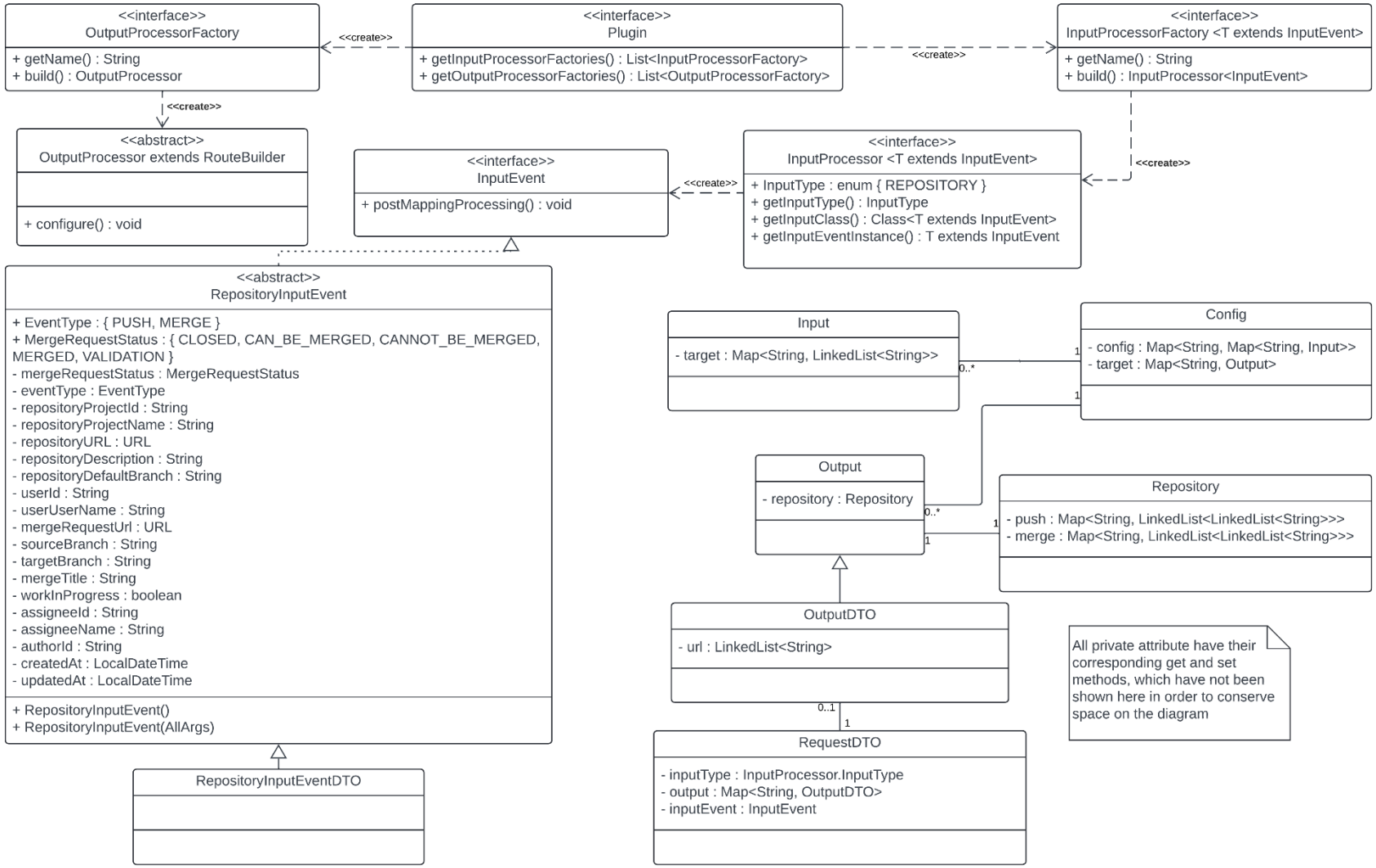
W celu zapewnienia możliwości dodania kolejnych obsługiwanych przez prototyp systemów zastosowana została architektura *pluginowa*. W tym celu zostało wykorzystane podejście *Service Provider Interface* (SPI), zaimplementowane w ramach projektu request-adapter-spi, definiującego interfejsy i klasy implementowane przez dostarczane wtyczki. Przygotowany został też pakiet request-adapter-core odpowiedzialny za wczytywanie rozszerzeń oraz przetwarzanie zapytań przychodzących do systemu. Na potrzeby prezentacji działania prototypu zostały przygotowane trzy wtyczki. Dwa z nich obsługują dane wejściowe z systemów GitLab i GitHub a jeden obsługuje wyjście dla Jiry. Na Rysunek 7 przedstawiony został poglądowy schemat przygotowanego prototypu.



Rysunek 7. Diagram komponentów systemu

5.1. Request Adapter SPI

Projekt zawierający interfejsy, definiujące strukturę wtyczek obsługiwanych przez przygotowany prototyp oraz klasy do implementacji przez te rozszerzenia. Na potrzeby modelu przygotowane zostały klasy pozwalające na obsługę danych wejściowych o typie repozytorium gitowego. Rysunek 8 przedstawia klasy dostarczone w ramach projektu oraz ich metody.



Rysunek 8. Diagram klas Request Adapter SPI

5.1.1 Plugin

Listing 5. Struktura interfejsu Plugin

```
public interface Plugin {
    default List<InputProcessorFactory> getInputProcessorFactories(){
        return Collections.emptyList();
    };
    default List<OutputProcessorFactory> getOutputProcessorFactories(){
        return Collections.emptyList();
    }
}
```

Interfejs przedstawiony na Listing 5 zawiera definicje list implementowanych *pluginów* wejściowych i wyjściowych. Każda wtyczka stworzona w ramach jego implementacji musi zagwarantować, że jej funkcjonalność będzie udostępniona w ramach przynajmniej jednej z tych dwóch list, aby została załadowana i wykorzystana w ramach projektu request-adapter-core z prototypu. Domyślnie obie listy są puste, aby zagwarantować, że program nie natrafi na problemy wynikające z wartości *NULL* na dowolnym etapie wykonania.

5.1.2 Input Plugin

Przedstawione w tej sekcji klasy definiują strukturę, którą musi zaimplementować każda wtyczka wejściowa, tak aby rozwiązania w niej dostarczone mogły być wykorzystane przez główny komponent. Możemy wyróżnić elementy przedstawione na Listing 6, Listing 7 i Listing 8:

Listing 6. Struktura interfejsu InputEvent

```
public interface InputEvent {
    public default void postMappingProcessing(){
        return;
    }
}
```

- *InputEvent* – interfejs dla wszystkich wydarzeń wejściowych, zawiera metodę *postMappingProcessing* pozwalającą wykonać dodatkowe mapowanie po wczytaniu całego wydarzenia. Wykorzystywana, jeżeli uzyskanie wszystkich niezbędnych danych nie było możliwe w oparciu tylko o dostarczoną strukturę.

Listing 7. Struktura interfejsu InputProcessor

```
public interface InputProcessor <T extends InputEvent>{
    public enum InputType { REPOSITORY }
    InputType getInputType();
    Class<T> getInputClass();
    T getInputEventInstance();
}
```

- *InputProcessor* – interfejs odpowiedzialny za rozpoczęcie procesowania wydarzenia wejściowego oraz dostarczenie informacji o klasie wydarzenia wejściowego dostarczanej przez wtyczkę,
    - *getInputType* – metoda pozwalająca nam zwrócić informacje o typie eventu wejściowego dla wybranej przez nas wtyczki (W ramach prototypu dostarczony jest tylko typ *REPOSITORY*),
    - *getInputClass* – metoda zwracająca informacje która klasa zaimplementowana przez wtyczkę implementuje *InputEvent*,
    - *getInputEventInstance* – metoda zwracająca nam nową instancję klasy implementującej *InputEvent*.

Listing 8. Struktura interfejsu InputProcessorFactory

```
public interface InputProcessorFactory {
    String getName();
    InputProcessor<InputEvent> build();
}
```

- *InputProcessorFactory* – interfejs definiujący dla jakiego systemu dostarczona została dana wtyczka, zwracający instancje zaimplementowanego procesora danych wejściowych.
    - *getName* – zwraca nazwę systemu, dla którego stworzony został dany *plugin*,
    - *build* – zwraca instancje zaimplementowanego przez wtyczkę procesora.

5.1.3 RepositoryInputEvent

Klasa implementująca *InputEvent* do której sprowadzone muszą być wszystkie rozszerzenia wejściowe typu repozytorium wczytane przez system. Definiuje zestaw pól, które są wykorzystywane w trakcie procesowania przez główny projekt systemu.

Listing 9. Struktura klasy RepositoryInputEvent

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public abstract class RepositoryInputEvent implements InputEvent {
    public enum EventType { PUSH, MERGE}
    public enum MergeRequestStatus {CLOSED, CAN_BE_MERGED,
CANNOT_BE_MERGED, MERGED, VALIDATION}
    private MergeRequestStatus mergeRequestStatus;
    private EventType eventType;
    private String repositoryProjectId;
    private String repositoryProjectName;
    private URL repositoryURL;
    private String repositoryDescription;
    private String repositoryDefaultBranch;
    private String userId;
    private String userUserName;
    private URL mergeRequestUrl;
    private String sourceBranch;
    private String targetBranch;
    private String mergeTitle;
    private boolean workInProgress;
    private String assigneeId;
    private String assigneeName;
    private String authorId;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
}
```

Wszystkie atrybuty zdefiniowane w klasie przedstawionej na Listing 9 posiadają odpowiadające im metody typu *Getter* i *Setter* automatycznie wygenerowane przy pomocy narzędzia Lombok poprzez adnotacje widoczne nad definicją klasy.

5.1.4 Properties

Zestaw klas opisujących strukturę obsługiwanych przez system plików YAML definiujących przepływy pomiędzy wtyczkami wejściowymi a wyjściowymi. Opis poszczególnych parametrów pliku konfiguracyjnego został umieszczony w paragrafie 6.1.

5.1.5 Output Plugin

Definiuje strukturę klas obsługujących systemy wyjściowe, wykorzystywane przez główny komponent prototypu request-adapter-core. Możemy wyróżnić następujące klasy, przedstawione w Listing 10 i Listing 11:

Listing 10. Struktura klasy abstrakcyjnej OutputPlugin

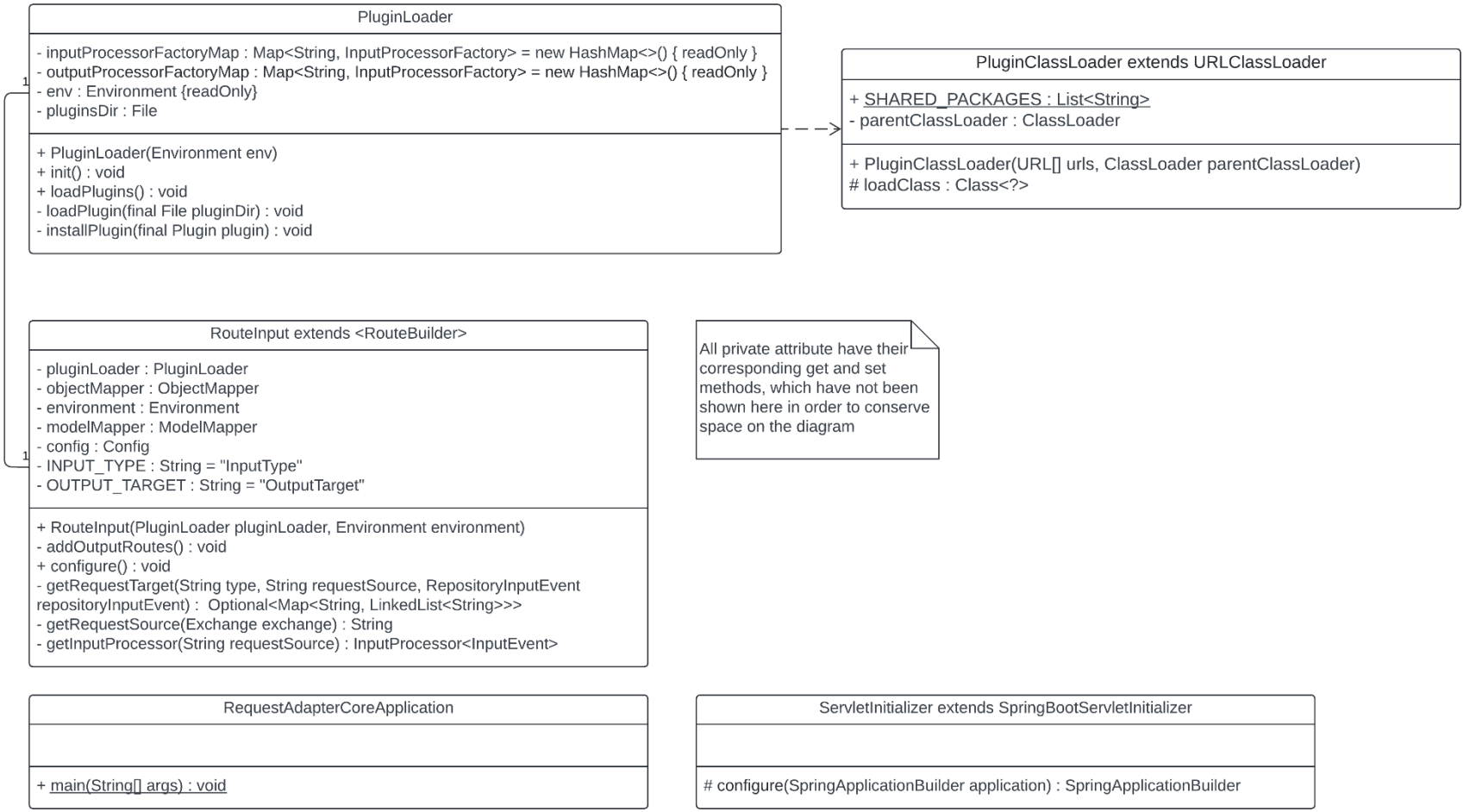
```
public abstract class OutputProcessor extends RouteBuilder {
    @Override
    public abstract void configure();
}
```

Abstrakcyjna klasa rozszerzająca definicje *RouteBuilder* dostarczaną w pakiecie Apache Camel [1]. Pozwala na zdefiniowanie procesorów zapytań w metodzie *configure*, które będą następnie wykorzystywane w trakcie procesowania przepływów programu.

Listing 11. Struktura klasy OutputProcessorFactory

```
public interface OutputProcessorFactory {
    String getName();
    OutputProcessor build();
}
```

- *OutputProcessorFactory* – interfejs definiujący dla jakiego systemu wyjściowego dostarczona została dana wtyczka, zwracający instancje zaimplementowanego procesora danych wyjściowych,
    - *getName* – zwraca nazwę systemu, dla którego stworzony została dana wtyczka,
    - *build* – zwraca instancje zaimplementowanego przez *plugin* procesora.



Rysunek 9. Diagram klas komponentu Request Adapter Core

5.2. Request Adapter Core

Główny komponent przygotowanego prototypu, dostarczony w formie aplikacji SpringBoot, mający za zadanie wczytać dostępne wtyczki, zainstalować dostarczone w nich klasy i udostępnić adres, na który trafiać będą informacje o wydarzeniach mających miejsce w systemach wejściowych. Na Rysunek 9 przedstawione zostały klasy dostarczone w ramach tego projektu wraz z opisem ich przeznaczenia.

5.2.1 ServletInitializer

Klasa rozszerzająca *SpringBootServletInitializer*, automatycznie wygenerowana przez IntelliJ w trakcie inicjalizacji projektu SpringBoot [1]. Przedstawiona na Listing 12.

Listing 12. Struktura klasy ServletInitializer

```
public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(RequestAdapterCoreApplication.class);
    }

}
```

5.2.2 PluginLoader

Klasa wykorzystywana do automatycznego wczytywania *pluginów* dostępnych w katalogu zdefiniowanym w pliku parametryzacyjnym Spring [12] w polu o nazwie *request.adapter.plugins.directory*. Odszukane w ten sposób wtyczki są następnie wczytywane przy pomocy klasy *PluginClassLoader*. W dalszej części tego paragrafu zostały przedstawione kluczowe metody klasy *PluginLoader*:

- *loadPlugins* – Rozpoczyna proces wczytywania wtyczek poprzez weryfikacje czy wskazana ścieżka istnieje i czy jest folderem a następnie wykonanie metody *loadPlugin* na każdym znalezionym pliku jar. Przedstawiona na Listing 13,

Listing 13. Struktura metody loadPlugins

```
public void loadPlugins() {
    if (!pluginsDir.exists() || !pluginsDir.isDirectory()) {
        System.err.println("Skipping Plugin Loading. Plugin dir not found:
" + pluginsDir);
        return;
    }

    final File[] files = requireNonNull(pluginsDir.listFiles());
    for (File pluginDir : files) {
        if (pluginDir.getName().endsWith("jar")) {
            loadPlugin(pluginDir);
        }
    }
}
```

- *loadPlugin* – Metoda przedstawiona na Listing 14, odpowiedzialna za wczytanie klas zawartych w dostarczonych wtyczkach do ścieżki wykonania programu. Wykorzystuję w tym celu klasę *PluginClassLoader*, której dokładny opis znaleźć można w następnej sekcji pracy,

Listing 14. Struktura metody loadPlugin

```
private void loadPlugin(final File pluginDir) {
    System.out.println("Loading plugin: " + pluginDir);
    URLClassLoader pluginClassLoader = null;
    try {
        pluginClassLoader = new URLClassLoader(
                new URL[]{pluginDir.toURI().toURL()},
                getClass().getClassLoader()
        );
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }
    final ClassLoader currentClassLoader =
Thread.currentThread().getContextClassLoader();
    try {
        Thread.currentThread().setContextClassLoader(pluginClassLoader);
        for (Plugin plugin : ServiceLoader.load(Plugin.class, pluginClassLoader)) {
            installPlugin(plugin);
        }
    } finally {
        Thread.currentThread().setContextClassLoader(currentClassLoader);
    }
}
```

- *installPlugin* – wywoływana z poziomu metody *loadPlugin*, dodaje zainstalowane rozszerzenia do map w celu umożliwienia łatwego do nich dostępu z poziomu klasy obsługującej zapytania przychodzące. Pokazana na Listing 15.

Listing 15. Struktura metody installPlugin

```
private void installPlugin(final Plugin plugin) {
    System.out.println("Installing plugin: " +
plugin.getClass().getName());
    for (InputProcessorFactory inputProcessorFactory :
plugin.getInputProcessorFactories()) {
        inputProcessorFactoryMap.put(inputProcessorFactory.getName(),
inputProcessorFactory);
    }
    for(OutputProcessorFactory outputProcessorFactory :
plugin.getOutputProcessorFactories()){
        outputProcessorFactoryMap.put(outputProcessorFactory.getName(),
outputProcessorFactory);
    }
}
```

5.2.3 PluginClassLoader

Klasa rozszerzająca *URLClassLoader*, pozwalająca na wczytanie klas zawartych w dostarczonych wtyczkach. Po wywołaniu z poziomu klasy *PluginLoader* następują weryfikacje mające na celu sprawdzenie czy dana klasa nie została już załadowana w ramach jednej z wcześniej już wczytanych paczek. Wczytywane klasy muszą mieć nazwę *package* zaczynającą się od *com.request.adapter.spi*. Na Listing 16 został przedstawiony kod kluczowej metody dostarczonej klasy:

Listing 16. Struktura metody loadClass

```
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException {

    Class<?> loadedClass = findLoadedClass(name);
    if (loadedClass == null) {
        final boolean isSharedClass =
SHARED_PACKAGES.stream().anyMatch(name::startsWith);
        if (isSharedClass) {
            loadedClass = parentClassLoader.loadClass(name);
        } else {
            loadedClass = super.loadClass(name, resolve);
        }
    }

    if (resolve) {
        resolveClass(loadedClass);
    }
    return loadedClass;
}
```

5.2.4 RouteInput

Klasa odpowiedzialna za przyjęcie zapytań przychodzących do systemu, zmapowanie ich zawartości do wspólnego schematu zdefiniowanego w module SPI i przekazanie ich do odpowiedniego *pluginu* wyjściowego w celu dalszego procesowania. Na Listing 17 i Listing 18 został przedstawiony opis metod wykorzystywanych przy przyjmowaniu zapytania z innego systemu.

Listing 17. Camel route przyjmujący zapytania do prototypu

```
rest("/adapter")
        .post("/input")
        .consumes(MediaType.APPLICATION_JSON_VALUE)
        .produces(MediaType.APPLICATION_JSON_VALUE)
        .to("direct:input");
```

Kod przedstawiony w Listing 17 przyjmuje zapytania post w formacie JSON pod adresem */adapter/input* a następnie przekazuje je do *route input* opisanego na Listing 18.

Listing 18. Struktura route input

```
from("direct:input")
        .process(exchange -> {
            String requestSource = this.getRequestSource(exchange);
            exchange.getIn().setHeader(
                    INPUT_TYPE,
                    this.pluginLoader.getInputProcessorFactoryMap()
                            .get(requestSource)
                            .build()
                            .getInputType()
            );
        }).toD("direct:${header." + INPUT_TYPE + "}");
```

W Listing 18 sprawdzane jest źródło pochodzenia danego zapytania i na jego podstawie wybierany jest typ systemu, dla którego będzie wykonywane dalsze procesowanie danych w prototypie. W obecnej formie prototypu dostępny jest typ wejścia *repository*.

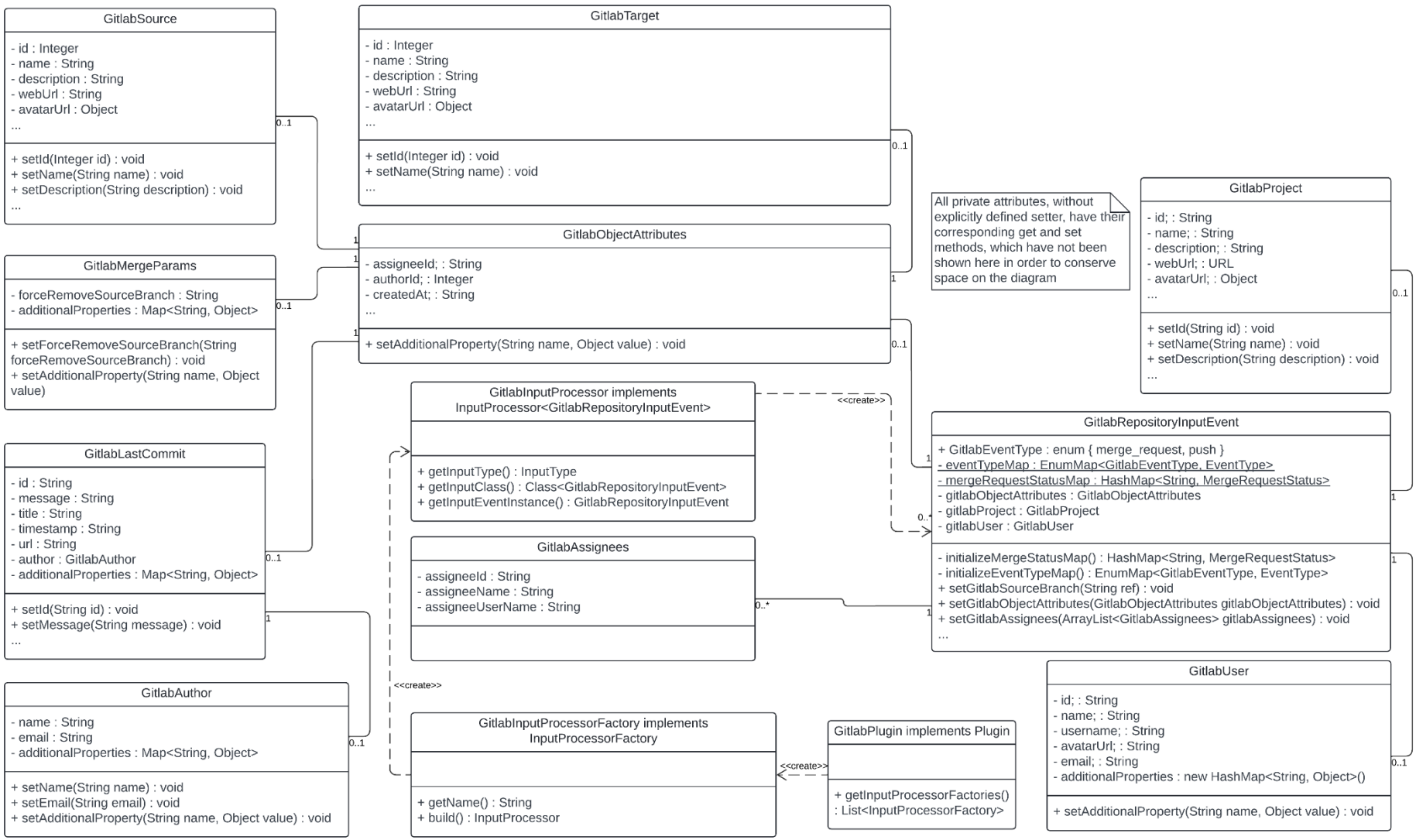
direct:REPOSITORY to główny procesor odpowiedzialny za zmapowanie wartości dostarczonych w zapytaniu wejściowym na wspólny model zdefiniowany w projekcie request-adapter-spi, przy pomocy dostarczonej wtyczki wejściowej. Następnie jest on odpowiedzialny za przekazanie tego obiektu do *route* dostarczonego w ramach odpowiedniej wtyczki wyjściowej. Kluczowe informacje o systemie wyjściowym system pobiera z dostarczonego pliku YAML czego przykład został przedstawiony na Listing 19.

Listing 19. Struktura metody getRequestTarget

```
private Optional<Map<String, LinkedList<String>>> getRequestTarget(String
type, String requestSource, RepositoryInputEvent repositoryInputEvent){

    if(requestSource == null)
        return Optional.empty();

    Map<String, Input> config = this.config.getConfig().getOrDefault(
            type,
            null
    );

    if(config == null)
        return Optional.empty();

    Input input = config.getOrDefault(
            repositoryInputEvent.getRepositoryURL().toString(),
            null
    );

    if(input == null)
        input = config.getOrDefault(type, null);

    if(input == null){
        input = config.getOrDefault("default", null);
        if(input == null)
            return Optional.empty();
    }
    return Optional.of(input.getTarget());
}
```

5.3. Plugin wejściowy GitLab

Na Rysunek 10 została przedstawiona przykładowa wtyczka obsługująca system GitLab, mająca za zadanie przekonwertować JSON przychodzący w zapytaniu wejściowym na wspólny format zdefiniowany w *request-adapter-spi* wykorzystując jackson-databind [13]. Na Listing 20 przedstawione zostało przykładowe mapowanie przychodzącego zapytania REST na domyślny format prototypu dla Systemu *repository* przy pomocy wtyczki GitLab. Ze względu na zakres zapytania w przykładzie skupimy się tylko na mapowaniu URL repozytorium do modelu.



Rysunek 10. Struktura pluginu GitLab

Listing 20. Przykładowa struktura sekcji project z zapytania API

```
"project": {
    "id": 19288925,
    "name": "TestProject",
    "description": "",
    "web_url": "https://gitlab.com/kaneckimaciej/testproject2",
    "avatar_url": null,
    "git_ssh_url": "git@gitlab.com:kaneckimaciej/testproject.git",
    "git_http_url": "https://gitlab.com/kaneckimaciej/testproject.git",
    "namespace": "Maciej Kanecki",
    "visibility_level": 0,
    "path_with_namespace": "kaneckimaciej/testproject",
    "default_branch": "master",
    "ci_config_path": "",
    "homepage": "https://gitlab.com/kaneckimaciej/testproject",
    "url": "git@gitlab.com:kaneckimaciej/testproject.git",
    "ssh_url": "git@gitlab.com:kaneckimaciej/testproject.git",
    "http_url": "https://gitlab.com/kaneckimaciej/testproject.git"
  },
```

Po otrzymaniu zapytania wraz z przedstawionym na Listing 20 fragmentem, klasa *RouteInput* dostarczona w projekcie *Request Adapter Core* automatycznie podejmie próbę mapowania zapytania na klasę *GitlabRepositoryInputEvent* dostarczoną w ramach pluginu Gitlab. Sekcja *project* jest w trakcie tej procedury automatycznie mapowana na klasę *GitlabProject*. Zmapowane wartości są następnie nanoszone na domyślny model repozytorium dostarczony w ramach projektu *request-adapter-spi*. Na Listing 22 i Listing 21 zamieszczone zostały fragmenty kodu wykorzystywane podczas opisanej procedury mapowania danych.

Listing 21. Metoda setGitlabProject z klasy GitlabRepositoryInputEvent

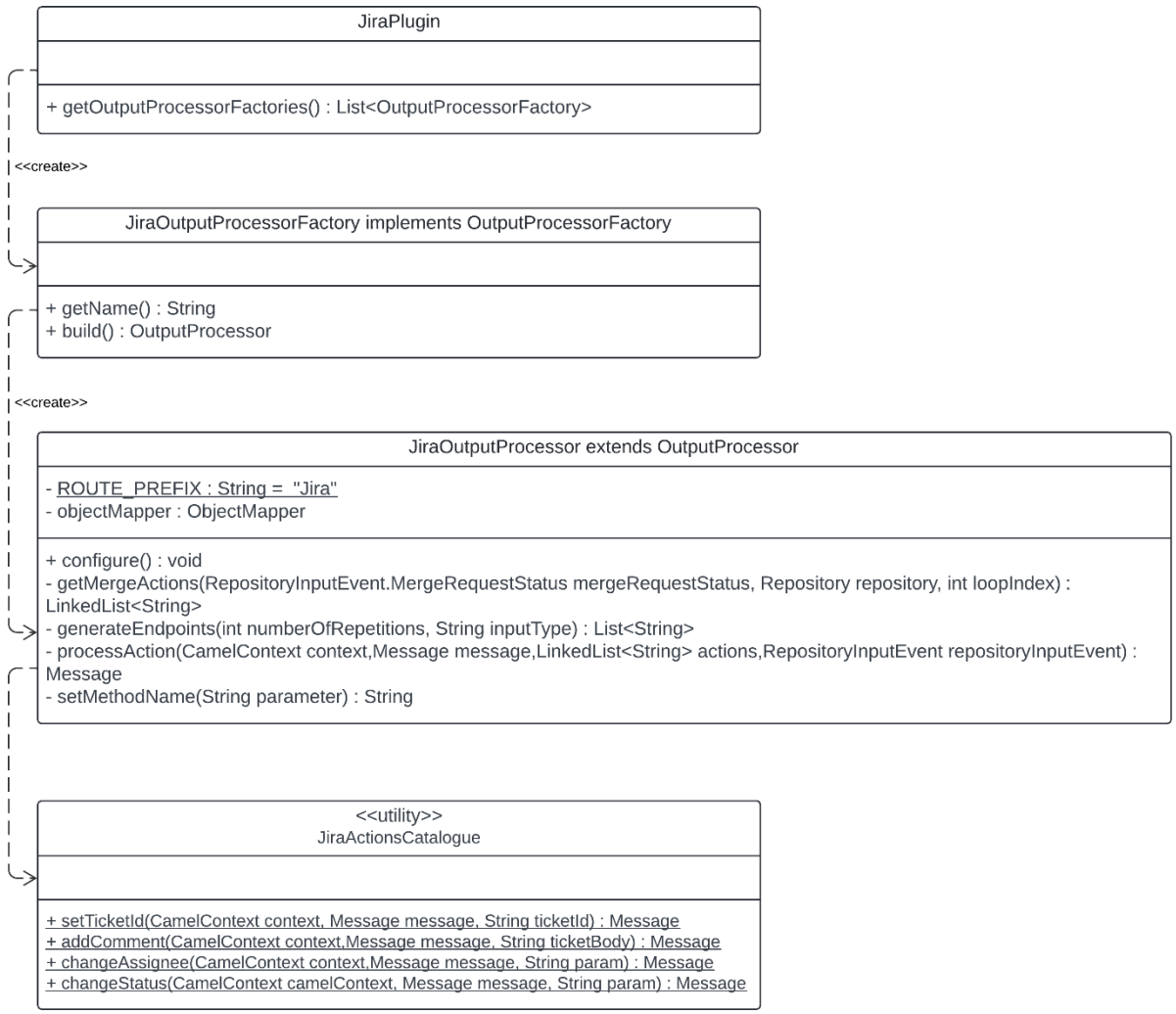
```
@JsonProperty("project")
public void setGitlabProject(GitlabProject gitlabProject) {
    this.gitlabProject = gitlabProject;
    super.setRepositoryProjectId(gitlabProject.getId());
    super.setRepositoryProjectName(gitlabProject.getName());
    super.setRepositoryURL(gitlabProject.getWebUrl());
    super.setRepositoryDescription(gitlabProject.getDescription());
    super.setRepositoryDefaultBranch(gitlabProject.getDefaultBranch());
}
```

Listing 22. Fragment klasy GitlabProject

```
@Getter
@JsonIgnoreProperties(ignoreUnknown = true)
public class GitlabProject {

    @JsonProperty("id")
    private String id;
    @JsonProperty("name")
    private String name;
    @JsonProperty("description")
    private String description;
    @JsonProperty("web_url")
    private URL webUrl;
    @JsonProperty("avatar_url")
    private Object avatarUrl;
    @JsonProperty("git_ssh_url")
    private String gitSshUrl;
    @JsonProperty("git_http_url")
    private String gitHttpUrl;
    @JsonProperty("namespace")
    private String namespace;
    @JsonProperty("visibility_level")
    private Integer visibilityLevel;
    @JsonProperty("path_with_namespace")
    private String pathWithNamespace;
```

Po zmapowaniu w opisany na Listing 21 sposób wszystkich wartości dostępnych w podstawowej klasie *RepositoryInputEvent* dane dostarczone w ramach zapytania wejściowego będą dostępne do dalszego przetwarzania w klasie *RouteInput* i wtyczkach wyjściowych.

5.4. Wtyczka wyjściowa Jira



Rysunek 11. Struktura wtyczki Jira

Na Rysunek 11 przedstawiona została przykładowa wtyczka obsługująca system Jira, wykonująca akcje na przekazanych do *pluginu* danych z systemów wejściowych. W celu obsługi przychodzących zapytań dostarcza własne *route'y* camelowe [1] w których pobiera z konfiguracji YAML informacje o tym jakie akcje mają być podjęte dla danego typu systemu i zapytania. W kolejnych sekcjach opisany został przykładowy przebieg wykonania zapytania w pluginie oraz jedna z akcji.

Listing 23. Route wejściowy pluginu Jira

```
from("direct:" + ROUTE_PREFIX)
        .process(exchange -> {
            Message message = exchange.getIn();
            RequestDTO requestDTO = (RequestDTO) message.getBody();
            message.setHeader("inputType",
requestDTO.getInputType().name());
            exchange.getIn().removeHeaders("Camel*");
            List<String> outputs = requestDTO
                    .getOutput()
                    .get(ROUTE_PREFIX)
                    .getUrl()
                    .stream()
                    .map(
                            url -> "direct:" + ROUTE_PREFIX
                                    + "_"
                                    + message.getHeader("inputType")
                                    +"_PRE_PROCESS")
                    .collect(Collectors.toList());
            message.setHeader("outputs", outputs);
        })
        .multicast()
        .recipientList(header("outputs"))
        .aggregationStrategy((oldExchange, newExchange) -> {
            return newExchange;
        });
```

W Listing 23 *plugin* przyjmuje zapytanie przychodzące z głównej klasy programu *RouteInput* i weryfikuje z jakiego typu systemu pochodzi. Następnie, w oparciu o uzyskane w ten sposób informacje, przekierowuje zapytanie do odpowiedniego pre-procesora. W tym procesorze konieczna jest między innymi weryfikacja, ile wiadomości musimy wysłać ze względu na możliwość skonfigurowania wielu zapytań dla każdej wykonanej w systemie akcji. Na Listing 24 przedstawiony został fragment kodu weryfikujący, ile zapytań będzie musiało być przetworzone w następnym procesorze dla wydarzenia typu POST.

Listing 24. Ustawianie ilości zapytań do przeprocesowania na podstawie konfiguracji

```
case PUSH:
    message.setHeader("repeat", repository.getPush().get("default").size());
    message.setHeader(
            "endpointList",
            this.generateEndpoints(
                    repository.getPush().get("default").size(),
                    (String)message.getHeader("inputType")
            )
    );
    break;
```

W następnej kolejności dla każdego zapytania przy pomocy refleksji [14] zostaną wywołane pasujące do jego typu akcje, modyfikujące strukturę Camelowej [1] klasy *Exchange* tak by zawierała pełną strukturę zapytania REST które jest na koniec wysyłane do systemu docelowego. Na Listing 25 zamieszczony został fragment kodu przedstawiający sposób wywoływania akcji poprzez refleksje.

Listing 25. Struktura metody processAction

```
private Message processAction(
        CamelContext context,
        Message message,
        LinkedList<String> actions,
        RepositoryInputEvent
        ) throws NoSuchMethodException, InvocationTargetException,
IllegalAccessException {
    for(String function : actions){
        String[] parameters = function.split(",");
        List<Object> params = new ArrayList<>();
        params.add(context);
        params.add(message);
        for(int i =1; i < parameters.length; i++){
            if(parameters[i].matches("\".*\"")) {
                params.add(parameters[i].replace("\"", ""));
                continue;
            }
            params.add(
                    repositoryInputEvent
                            .getClass()
                            .getMethod(this.setMethodName(parameters[i]))
                            .invoke(repositoryInputEvent).toString()
            );
        }
        Method[] methods = JiraActionsCatalogue.class.getMethods();
        for(Method method : methods){
            if(method.getName().equals(parameters[0])) {
                method.invoke(null, params.toArray());
                break;
            }
        }
    }
    return message;
}
```

W ramach prototypu rozszerzenia obsługującego system wyjściowy Jira zdefiniowane zostały 4 akcje:

- *setTicketId(ticketId)* – Akcja ustawiająca identyfikator zadania wykorzystywanego w kolejnych akcjach. Ustawiana w YAML wartość dynamicznie wyciągnięta z wspólnego modelu danych lub statycznie ustawiona na poprzez przekazanie wartości w cudzysłowie,
- *addComment(commentBody)* – Akcja dodająca komentarz do modyfikowanego zadania. Ustawiana wartość może być dynamiczna, poprzez podstawienie wartości z dostarczonego modelu danych lub statyczna,
- *changeAssignee(assigneeId)* – Pozwala nam ustawić właściciela danego zadania na jira. W tym celu konieczne jest przekazanie identyfikatora jednoznacznie wskazującego na danego użytkownika, takiego jak na przykład adres email albo nazwa użytkownika,
- *changeStatus(statusId)* – Pozwala na zmianę statusu wybranego zadania, wymaga przekazania id danego statusu jako parametr w pliku YAML.

6. Przykładowa konfiguracja i wykorzystanie systemu

W tym rozdziale przedstawione zostały przykłady wykorzystania systemu w realnym scenariuszu biznesowym wraz z jego konfiguracją dla systemów wejściowych typu *repository* i wyjściowego typu Jira oraz testami rozwiązania.

6.1. Opis pliku konfiguracyjnego

Modyfikacja logiki przygotowanego prototypu jest wykonywana poprzez zmianę parametrów pliku konfiguracyjnego YAML. W kolejnych podrozdziałach opisane zostały poszczególne sekcje tego szablonu wraz z przykładami, a na Listing 26 i Listing 27 przedstawiony został już w pełni skonfigurowany plik.

6.1.1 Sekcja config

Listing 26. Sekcja config przykładowego pliku YAML

```
config:
  repository:
    default:
      target:
        Jira:
          - default
    "https://gitlab.com/kaneckimaciej/testproject":
      target:
        Jira:
          - https://masters2-jira.atlassian.net/rest/api/2
    "https://github.com/Mac136/TestRepository2":
      target:
        Jira:
          - https://masters2-jira.atlassian.net/rest/api/2
```

Służy do definiowania mapowań pomiędzy różnymi typami systemów wyjściowych i wejściowych. W celu poprawnej konfiguracji narzędzia należy w niej wprowadzić nazwę typu konfigurowanego rozwiązania. W ramach przygotowanego prototypu opracowana została obsługa systemów wejściowych typu *repository* i sekcja o tej właśnie nazwie została przedstawiona na Listing 27. W pod sekcji repository definiowane są powiązania pomiędzy wybranym systemem wejściowym a wyjściowym. W celu zapewnienia możliwe maksymalnej elastyczności konfiguracji nowe systemy wejściowe dodawać można w następujące sposoby:

- *default* - pozwala zdefiniować domyślną opcje która będzie wykorzystywana, jeżeli dla danego *pluginu* wejściowego nie zostanie odnaleziona żadna bardziej specyficzna definicja,
- Nazwa systemu wejściowego (na przykład GitLab) – pozwala na wskazanie konkretnego systemu, z którego wszystkie zapytania będą przekierowywane do wybranych miejsc docelowych,
- URL Systemu wejściowego – umożliwia wskazanie konkretnego adresu systemu wejściowego z którego zapytania mają być kierowane do dalej zdefiniowanych systemów.

Dla każdego z uprzednio wskazanych systemów wejściowych zdefiniować możemy listę systemów wyjściowych do jakich przekierowywane mają być przychodzące zapytania. W ramach każdego typu platformy zdefiniować możemy dowolną ilość instancji tego systemu do jakich kierowane mają być zapytania. Dla każdej platformy wymagana jest dedykowana wtyczka wyjściowa do obsługi. W ramach przygotowanego prototypu opracowana została wtyczka dla systemu Jira.

6.1.2 Sekcja target

Listing 27. Sekcja target z przykładowego pliku YAML

```
target:
  Jira:
    repository:
      push:
        default:
          -
            - setTicketId,sourceBranch
            - addComment,sourceBranch
          -
            - setTicketId,sourceBranch
            - changeAssignee,userUserName
          -
            - setTicketId,sourceBranch
            - changeStatus,"21"
      merge:
        CLOSED:
          -
            - setTicketId,sourceBranch
            - changeAssignee,assigneeName
          -
            - setTicketId,sourceBranch
            - changeStatus,"31"
        CAN_BE_MERGED:
          -
            - setTicketId,sourceBranch
            - changeAssignee,assigneeName
          -
            - setTicketId,sourceBranch
            - changeStatus,"31"
        CANNOT_BE_MERGED:
          -
            - setTicketId,sourceBranch
            - changeAssignee,assigneeName
          -
            - setTicketId,sourceBranch
            - changeStatus,"31"
        MERGED:
          -
            - setTicketId,sourceBranch
            - changeAssignee,assigneeName
          -
            - setTicketId,sourceBranch
            - changeStatus,"31"
```

Przechowuje listę systemów wyjściowych wraz ze zdefiniowanymi dla nich przepływami. Na potrzeby tego przykładu skupimy się na wtyczce dostarczonej wraz z prototypem, czyli systemie Jira. Wykorzystamy też uprzednio skonfigurowany typ systemu wejściowego, czyli typ *repository*. Akcje udostępniane przez system Jira przedstawione zostały w sekcji 5.4 pracy. Dla typu *repository* możliwe jest skonfigurowanie następujących przepływów:

- *push* – przechowuje definicje przepływów dla wydarzeń typu *push* w systemie źródłowym. Dla tego typu wydarzenia możliwy jest tylko wybór opcji *default*,
- *merge* – pozwala na zdefiniowanie przepływów dla wydarzenia typu *merge* w repozytorium. W celu lepszego dopasowani definicji do typu zdarzenia, wyróżnić możemy następujące stany tego zdarzenia:
    - CLOSED – sygnalizuje, że powiązany *merge request* został zamknięty,
    - CAN_BE_MERGED – informuje nas o tym, że *merge request* przeszedł podstawową walidacje i może być zaakceptowany,
    - CANNOT_BE_MERGED – status wydarzenia w sytuacji, kiedy powiązany *merge request* nie może zostać zaakceptowany,
    - MERGED – status wydarzenia w momencie, kiedy *merge request* zostanie zaakceptowany.

6.2. Opis procesu

W tej sekcji przedstawiona została przykładowa seria problemów jakie mógłby wystąpić w firmie zajmującej się pracą programistyczną w trakcie interakcji z przedstawionym w dalszej części paragrafu uproszczonym procesem wytwarzania oprogramowania. W ten sposób zdefiniowane problemy postaramy się następnie rozwiązać przy pomocy dostarczonego prototypu w celu zobrazowania jaki potencjalnie wpływ mógłby wywrzeć na sposób działania organizacji.

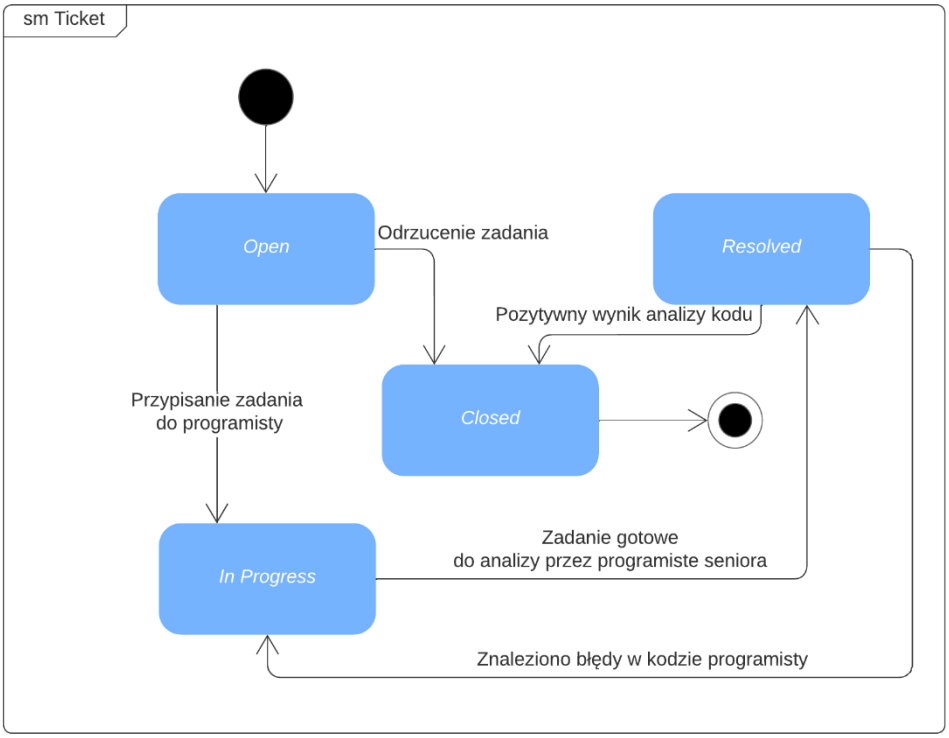
W omawianym uproszczonym procesie wyróżnić możemy następujące systemy:

- GitLab – główne repozytorium Git wykorzystywane głównie przez developerów,
- Jira – narzędzie kontroli nad przebiegiem realizacją zadań przez programistów. Aktualizowane przez developerów, ale wykorzystywane głównie przez kadrę menadżerską firmy w celu dystrybucji zadań oraz walidacji postępów.

W opisywanym procesie wyróżnić możemy dwóch uczestników zajmujących się wymienionymi na liście zadaniami:

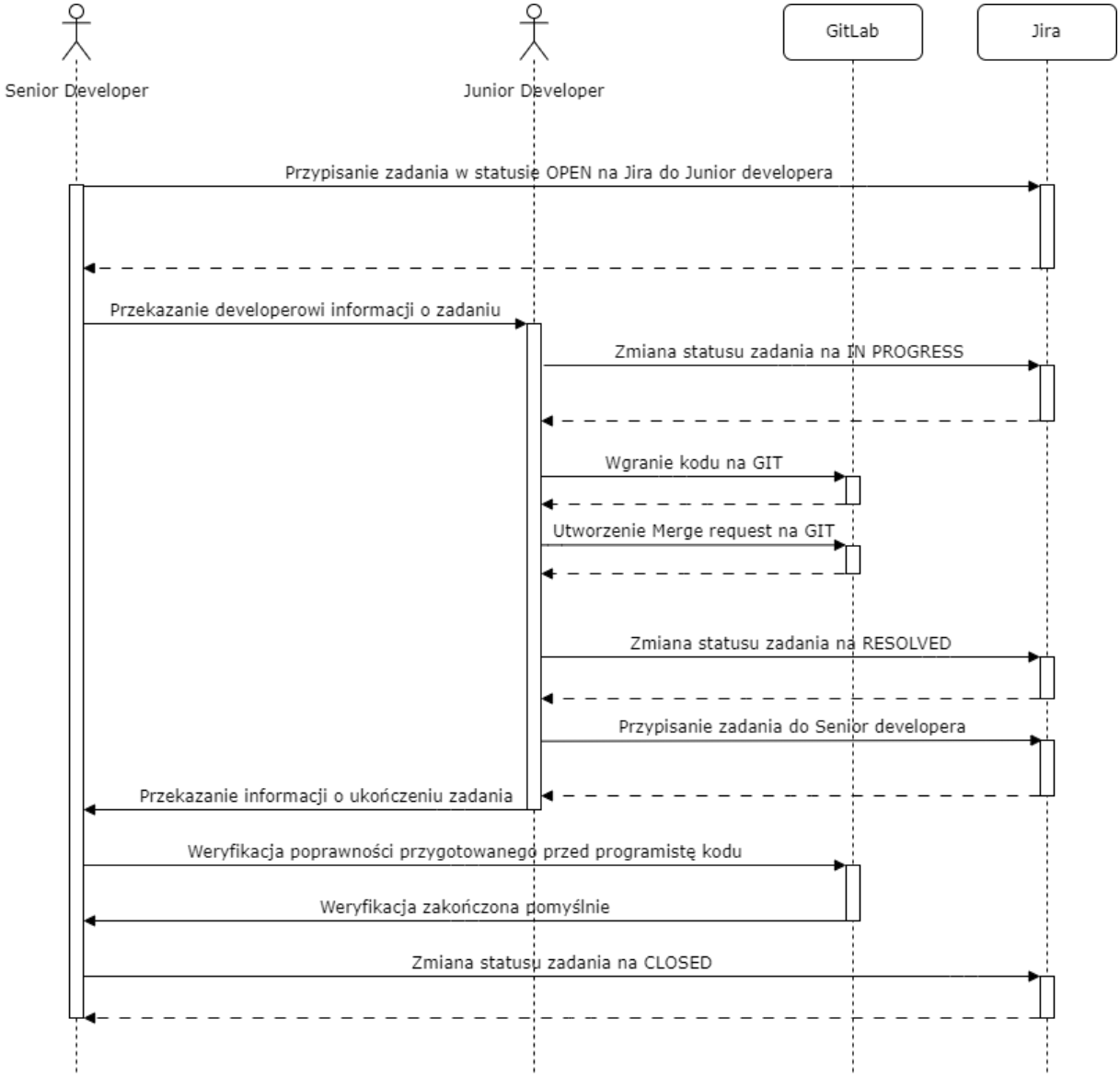
1. Programistę Juniora → Przygotowuje kod dostarczanej funkcjonalności,
2. Programista Senior → Weryfikuje poprawność dostarczanego przez Juniora kodu.

Każdemu fragmentowi cyklu życia naszego programu odpowiadać będzie dedykowany status na Jirze, której konfiguracja wykorzystywana w przykładowym przepływie programu nie powinna być uznawana za optymalną. Służy ona jedynie prezentacji kluczowych założeń prototypu i nie jest przedmiotem pracy. Statusy dostępne w domyślnych ustawieniach jakie są wykorzystywane w tym procesie zostały opisane na Rysunek 12.



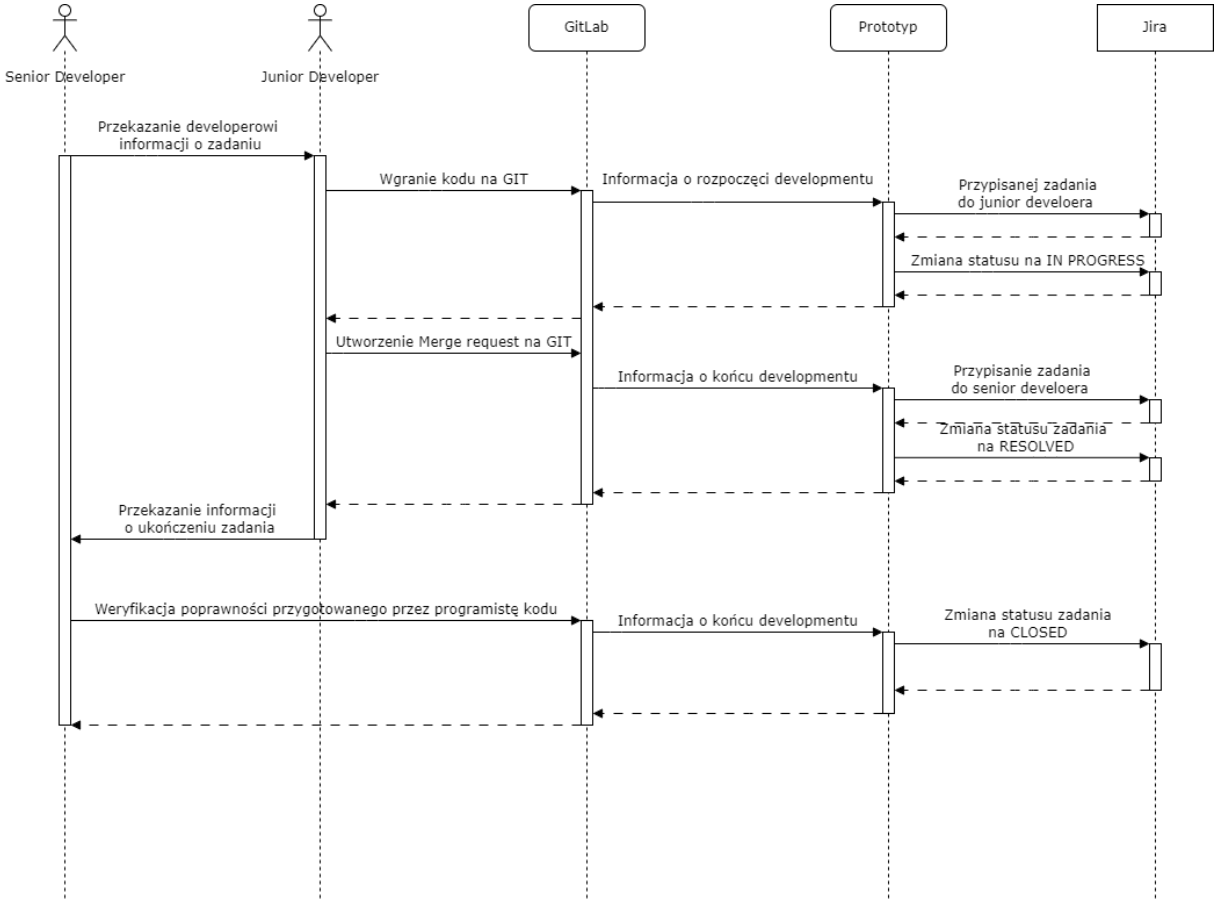
Rysunek 12. Przykładowy przepływ statusów Jira

- OPEN – Pierwszy status naszego zadania, w opisywanym procesie informujący nas o tym, że praca nad zadaniem jeszcze się nie zaczęła,
- IN PROGRESS – Status informujący nas o tym, że zadanie jest obecnie wykonywane,
- RESOLVED – Status informujący nas o tym, że developer skończył swoją pracę nad zadaniem i czeka na sprawdzenie jego rezultatów przez osobę bardziej doświadczoną,
- CLOSED – Status wskazujący na to, że praca nad danym zadaniem została sprawdzona przez programistów z większym doświadczeniem i zadanie możemy uznać za skończone.



Rysunek 13. Diagram sekwencji dla bezbłędnego przepływu procesu

Rysunek 13 przedstawia optymistyczny przepływ całego procesu, w którym nie napotkaliśmy trudności na żadnym jego etapie. Zauważyć tutaj można dużą ilość interakcji pomiędzy członkami zespołu a narzędziami wykorzystywanymi w ramach obsługi procesu i całkowity jej brak pomiędzy narzędziami. Oznacza to, że cały cykl wymaga dużego zaangażowania ze strony developerów w celu utrzymania aktualnego stanu Jiry i potencjalnie może prowadzić do błędów, jeżeli programista zapomni o konieczności zaktualizowania statusu zadania. Problemu tego można się pozbyć przy pomocy przygotowanego prototypu co zostało przedstawione na diagramie z Rysunek 14.



Rysunek 14. Diagram sekwencji dla bezbłędnego procesu z wykorzystaniem prototypu

Jak widać na Rysunek 14, dołączenie prototypu do procesu pozwoliło nam znacząco zredukować ilość pracy jaką programiści musieli poświęcić na aktualizacje statusów i przepisywanie zadań na innych członków zespołu. Pozwoliło im to w pełni skupić się na pracy programistycznej i zredukowało ilość potencjalnych pomyłek jakie mogłyby wystąpić w procesie poprzez usunięcie czynnika ludzkiego.

6.3. Test systemu

W tej sekcji przedstawiona została konfiguracja pliku YAML wymagana do osiągnięcia celów wyznaczonych w procesie opisanym w sekcji 6.1, wraz z listą kroków wymaganych do wykonania czynności opisanych w diagramie sekwencyjnym.

6.3.1 Konfiguracja YAML

Przed rozpoczęciem testów prototypu musimy upewnić się, że posiadany przez nas plik konfiguracyjny YAML został odpowiednio zdefiniowany. Zacząć musimy od sprawdzenia definicji mapowania systemów wejściowych na wyjściowe.

Listing 28. Przykład konfiguracji mapowania wejścia-wyjścia

```
config:
  repository:
    default:
      target:
        Jira:
          - https://masters-jira.atlassian.net/rest/api/2
```

Na przykładzie przedstawionym w Listing 28 zamieszczona została jedna z najprostszych możliwych konfiguracji, wejście dla każdego typu repozytorium zostało efektywnie przekierowane do wyjścia Jira pod podanym adresem IP.

Listing 29. Przykładowa konfiguracja przepływów programu

```
target:
  Jira:
    repository:
      push:
        default:
          -
            - setTicketId,sourceBranch
            - changeAssignee,userUserName
          -
            - setTicketId,sourceBranch
            - changeStatus,"4"
      merge:
        CAN_BE_MERGED:
          -
            - setTicketId,sourceBranch
            - changeAssignee,assigneeName
          -
            - setTicketId,sourceBranch
            - changeStatus,"5"
        MERGED:
          -
            - setTicketId,sourceBranch
            - changeStatus,"6"
          -
            - setTicketId,sourceBranch
            - addComment,"Task has been Reviewed & Approved by a developer"
```

Listing 29 przedstawia konfiguracje akcji pozwalającą nam odtworzyć proces opisany na wcześniej pokazanym diagramie sekwencji z wykorzystaniem prototypu (Rysunek 14). W momencie wykrycia przychodzącego wydarzenia *push* prototyp podejmie dwie akcje:

- Zmieniony zostanie właściciel zadania na autora akcji na Git,
- Status *ticketu* zmieniony zostanie na inny o numerze id 4, odpowiadający wartości IN PROGRESS.

W momencie, kiedy wykryty zostanie *Merge request* bez wad technicznych, zostaną przez system podjęte następujące akcje:

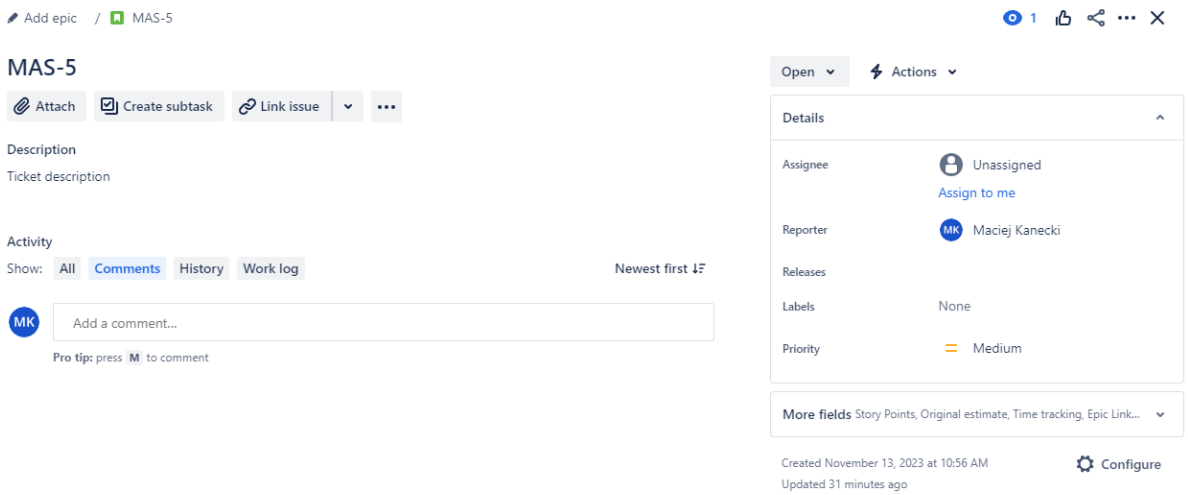
- Zadanie zostanie przepisane na osobę odpowiedzialną za recenzję tego *merge requesta*,
- Status na Jirze zostanie zmieniony na wartość RESOLVED o identyfikatorze 5.

Po akceptacji dostarczonego przez developera kodu i wywołaniu wydarzenia MERGED wykonane zostaną następujące akcje:

- Status zadania jest zmieniany na wartość CLOSED,
- Pod zadaniem pozostawiany jest komentarz, że zadanie to zostało zweryfikowane przez jednego z członków zespołu i nie zostały w nim znalezione żadne problemy.

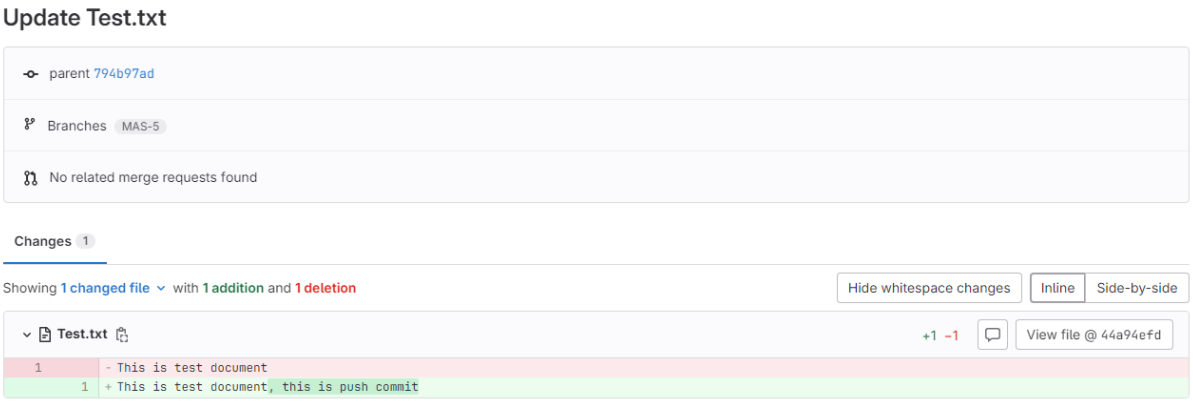
6.3.2 Wykonanie akcji zdefiniowanych w prototypie

W ramach testów naszego systemu skupimy się na cyklu życia jednego zadania o identyfikatorze MAS-5. Na początku naszego procesu zadanie to znajduje się w statusie OPEN i nie ma przypisanego żadnego właściciela. Jego stan zobaczyć możemy na Rysunek 15



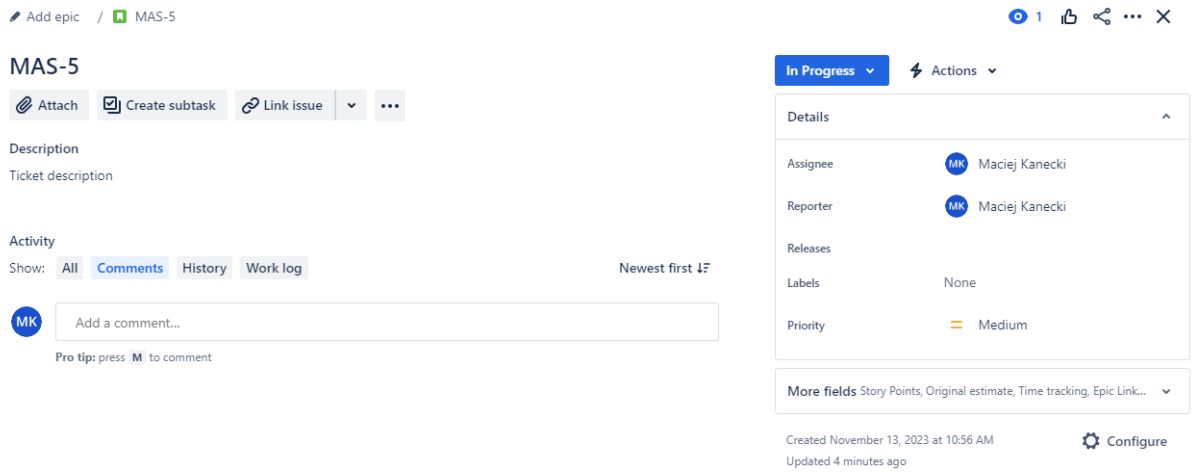
Rysunek 15. Przykład zadania MAS-5

Następnie programista junior rozpocznie wprowadzanie swoich zmian na gałęzi GIT [15] o nazwie MAS-5. Na potrzeby tego testu zmiana w kodzie aplikacji zasymulowana została poprzez edycje pliku tekstowego dostępnego na repozytorium i dodanie w nim zdania „*this is push commit*”, co zostało przedstawione na Rysunek 16.



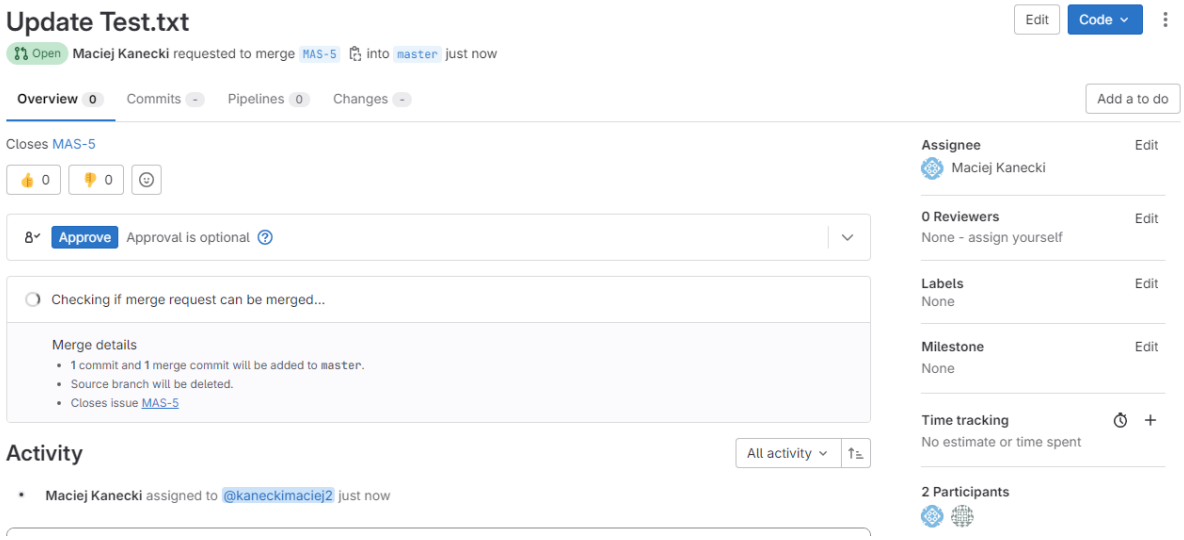
Rysunek 16. Zmiana zawartości pliku testowego w repozytorium

Po wykonaniu akcji z Rysunek 16 zadanie w Jira zostaje natychmiast przypisane do programisty, który wykonał zmianę a jego status zmienia się na IN PROGRESS (Rysunek 17).



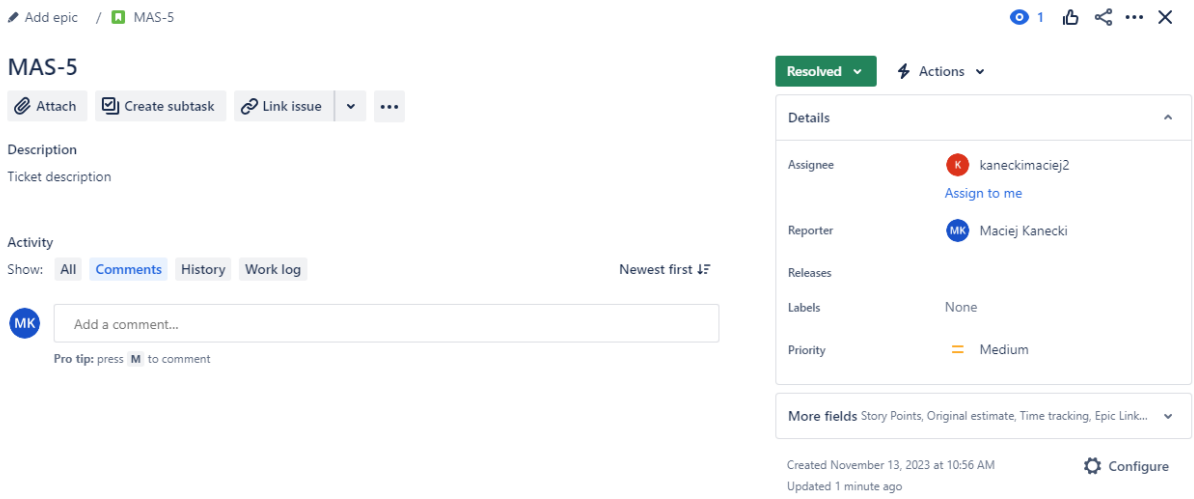
Rysunek 17. Zaktualizowane zadanie

Po ukończeniu pracy nad tym zadaniem programista musi przygotować *Merge Request* i przekazać swój kod do recenzji bardziej doświadczonemu współpracownikowi. Przykładowa konfiguracja takiego *Merge Requesta* przedstawiona została na Rysunek 18.



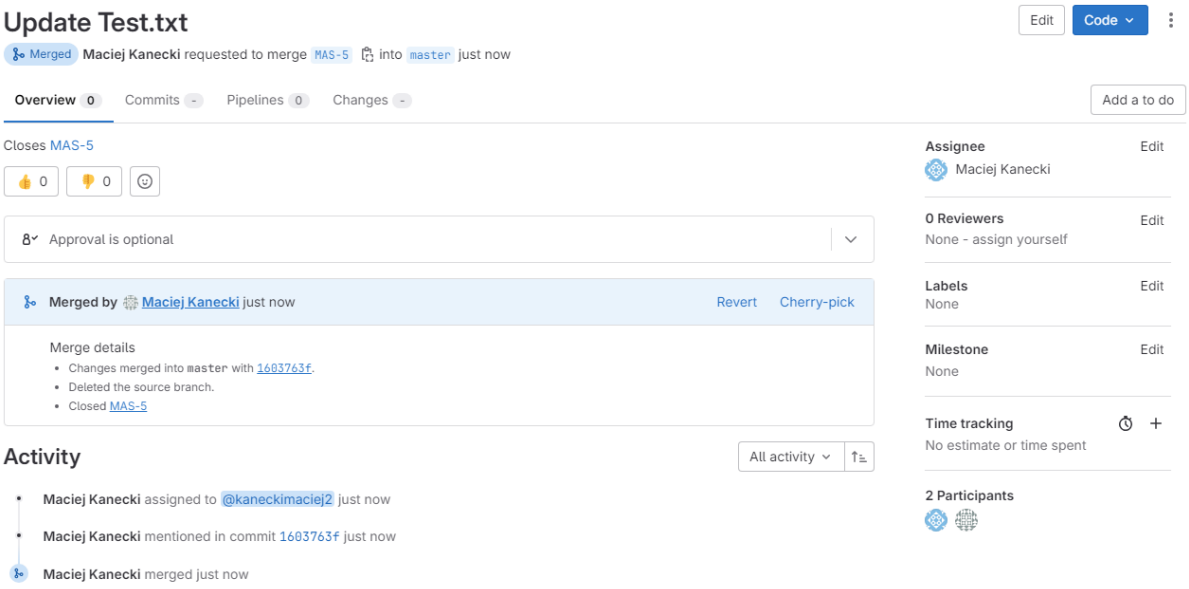
Rysunek 18. Merge Request założony przez programistę

Po utworzeniu przedstawionego na Rysunek 18 *Merge Request*, prototyp automatycznie zaktualizuje Jire poprzez zmianę statusu zadania na RESOLVED i zmianę właściciela zadania na developera odpowiedzialnego za recenzje jakości kodu (Rysunek 19).

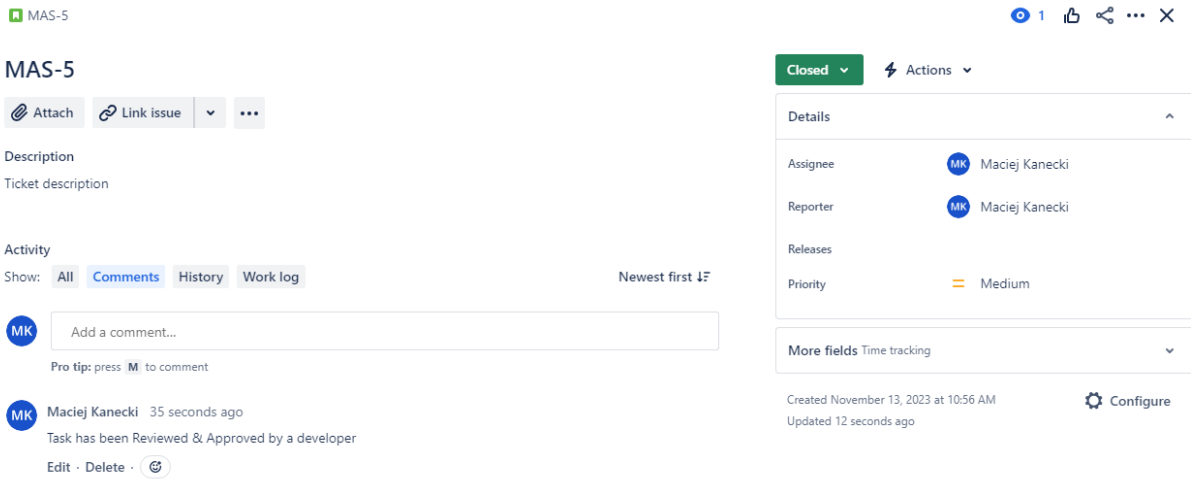


Rysunek 19. Stan zadania po utworzeniu Merge Request

Następnym krokiem jest recenzja kodu i weryfikacja poprawności założonego *Merge Requesta*. Po pomyślnej weryfikacji przez bardziej doświadczonego developera *merge request* jest następnie akceptowany, co zostało przedstawione na Rysunek 20, a informacja o zmianie statusu zadania jest wysyłana do skonfigurowanej instancji Jira (Rysunek 21).



Rysunek 20. Zaakceptowany Merge Request



Rysunek 21. Zadanie na Jira odzwierciedlające zmiany na GitLab

7. Podsumowanie

W przedstawionej pracy przeanalizowany zostały problemy z synchronizacją narzędzi wykorzystywanych na projektach z jakimi mierzą się programiści w trakcie swojej pracy. Posiłkując się innymi badaniami z tej dziedziny przeprowadzonymi na przestrzeni ostatnich lat, przedstawiona została skala tego problemu i potencjalne zyski jakie jego rozwiązanie mogłoby przynosić przedsiębiorstwom operującym na rynku.

Opisane zostały też konkurencyjne, obecnie dostępne, rozwiązania i przedstawione zostały ich wady i zalety. Analiza ta wykazała istnienie pewnego zakresu funkcji który nie jest obecnie adresowany przez żadne z nich. Wskazuje nam to na pewną lukę na rynku, która mogłaby być wykorzystana przez bardziej generyczne rozwiązanie.

W sekcji 5 opisane zostało generyczne narzędzie pozwalające część zidentyfikowanych w trakcie analizy funkcji zaimplementować. Przygotowany prototyp dostarcza podstawowe funkcjonalności pozwalające na integracje między systemami i wykonanie standardowych akcji w systemie Jira, ale pozostawia też miejsce na dalszy rozwój dzięki architekturze opartej na wtyczkach.

W kolejnych krokach rozwoju prototypu należałoby się skupić na poszerzaniu zakresu akcji jakie mogą być podjęte w istniejącym pluginie Jira oraz na dostarczeniu kolejnych wtyczek pozwalających na integracje z większym zakresem systemów. Po podjęciu opisanych w sekcji 6.3 kroków, zaproponowane rozwiązanie pozwalałoby uprościć dużą część czynności, które regularnie muszą wykonywać developerzy w trakcie pracy na projektach informatycznych. Prowadziłoby to do redukcji kosztów co mogłoby pozytywnie wpłynąć na zyskowność projektu.

8. Prace cytowane i źródła wiedzy

[1] How Much Time Do Developers Spend Actually Writing Code? (Dostęp 28.01.2024) https://thenewstack.io/how-much-time-do-developers-spend-actually-writing-code/
[2] Today was a Good Day: The Daily Life of Software Developers, (Dostęp 28.01.2024) https://www.microsoft.com/en-us/research/uploads/prod/2019/04/devtime-preprint-TSE19.pdf
[3] State of the java ecosystem, (Dostęp 22.05.2024) https://newrelic.com/resources/report/2023-state-of-the-java-ecosystem
[4] Most Popular Java IDEs in 2024, (Dostęp 26.05.2024) https://www.jrebel.com/blog/best-java-ide
[5] Lombok, (Dostęp 26.05.2024) https://projectlombok.org/
[6] Lombok @Data tag dokumentacja, (Dostęp 26.05.2024) https://projectlombok.org/features/Data
[7] Spring Initializr, (Dostęp 28.04.2024) https://start.spring.io/
[8] Gradle documentation, (Dostęp 28.04.2024) https://docs.gradle.org/current/userguide/userguide.html
[9] Gradle vs Maven, (Dostęp 28.04.2024) https://gradle.org/maven-vs-gradle/
[10] Gradle vs. Maven: Performance, Compatibility, Builds, & More, (Dostęp 28.04.2024) https://stackify.com/gradle-vs-maven
[11] Apache Camel, (Dostęp: 27.01.2024) https://camel.apache.org/
[12] Spring Boot Reference Documentation, (Dostęp: 27.01.2024) https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
[13] Jackson-Databind, (Dostęp 27.01.2024) https://github.com/FasterXML/jackson-databind
[14] Java Reflection, (Dostęp 28.01.2024) https://www.oracle.com/technical-resources/articles/java/javareflection.html
[15] Git Branching, (Dostęp 30.01.2024) https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
[16] Podstawowe informacje o Jira, (Dostęp 09.05.2024) https://www.atlassian.com/software/jira/guides/getting-started/introduction#dig-into-specific-features
[17] Podstawowa wiedza o Git, (Dostęp 09.05.2024) https://en.wikipedia.org/wiki/Git
[18] Podstawowe informacje o narzędziu GitHub, (Dostęp 09.05.2024) https://en.wikipedia.org/wiki/GitHub
[19] Podstawowe informacje o narzędziu Bitbucket. (Dostęp 09.05.2024) https://en.wikipedia.org/wiki/Bitbucket
[20] „Introduction to Spring Framework”, (Dostęp 11.05.2024) https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/overview.html
[21] Dokumentacja CamelContext, (Dostęp 11.05.2024) https://camel.apache.org/manual/camelcontext.html

9. Spis tabel

10. Spis rysunków

11. Spis listingów