TREŚĆ

WPROWADZENIE…………….……………………………………………...3

1. ARCHITEKTURA I TECHNOLOGIE PROGRAMU TESTOWEGO .........................................................................................................................4

2. WYMAGANE DANE W POSTACI MODELI DO TESTOWANIA KOLEKCJI KART ............... .............................. ........................................11

2.1 Modele w katalogu kart/katalogu ..................... .....................................11

2.2 Modele w katalogu karta/klient ..................... ........................................13

3. ZBIÓR TESTÓW ........................ .............................................................15

3.1 Zwykłe testy ........................ .............................. ....................................15

3.1.1. RedundantTrieTest ........................ .............................. ......................15

3.1.2. TestResourceGenerator ..................... .............................. ..................15

3.1.3. MtgTestApplicationTests ..................... ............................................ .16

3.2 Testy integracyjne ........................ ........................................................ .16

3.2.1. MagicTheGatheringCardCatalogIntegrationTest ...............................16

3.2.2. MagicTheGatheringCardClientIntegrationTest ..................................18

3.2.3. CardSearchTrieIntegrationTest ...........................................................19

3.3 Testy jednostkowe ........................... ......................................................19

3.3.1. MagicTheGatheringCardCatalogUnitTest .........................................19

3.3.2. CardSearchTrieUnitTest ..................... ...............................................21

3.3.3. RedundantTrieUnitTest ..................... ................................................22

PODSUMOWANIE ........................... .........................................................24

LISTA WYKORZYSTANYCH ŹRÓDEŁ..................................................25

**WPROWADZENIE**

REST-Assured to biblioteka Java, która zapewnia język specyficzny dla domeny (DSL) do pisania wydajnych, łatwych w utrzymaniu testów dla interfejsów API RESTful. Jedną z rzeczy, którą naprawdę lubię w rest-assured, jest styl pisania testów BDD, dzięki czemu można bardzo wygodnie czytać testy, co dzieje się w środku i jak będą przebiegać testy.

Celem tej pracy jest skonstruowanie modułu testowego przy założeniu, że:

1) Dokumentacja API jest dostępna pod adresem: https://docs.magicthegathering.io/.

2) Wszystkie dostępne punkty końcowe będą objęte testami.

3) Testy będą zawierały scenariusze pozytywne (sprawdzanie logiki, zwracanej treści) i negatywne (sprawdzanie kodów błędów takich jak 404).

4) Testy będą wykorzystywać parametry (pathparam, param).

5) Klasy testów będą logicznie oddzielone.

6) Opisy/nazwy testów będą miały sens, tak aby użytkownik wiedział, co dany test powinien robić.

Aby zbadać i zbudować moduł testowy, użyjemy następujących interfejsów API - https://api.magicthegathering.io/v1/cards

**1. ARCHITEKTURA I TECHNOLOGIE PROGRAMU TESTOWEGO**Ten program zawiera reaktywnego klienta REST API udostępniającego informacje o kartach Magic: the Gathering. Celem jest umożliwienie konsumentom pobierania danych kart w postaci obiektu Java i umożliwienie im odczytywania stron danych zamiast polegania na synchronicznym procesie, który próbuje pobrać wszystkie ~ 10 tysięcy kart naraz.

Planowane ulepszenia klienta:

1) Pobieranie kart według nazwy lub identyfikatora zbieracza.

2) Więcej informacji o karcie, np. linki do zdjęć.

3) Pobieranie kart według legalności formatu, np. Standard, Modern, Commander.

Ostatecznym celem tego klienta jest wykorzystanie go przez usługę REST, która może służyć do przetwarzania aplikacji internetowych.

Program ten składa się z modeli różnych kart w katalogu i operacji z nimi za pomocą klienta, konfiguracji i wyszukiwania oraz, oczywiście, kilku różnych rodzajów testów, takich jak zwykłe, integracyjne i jednostkowe.

Teraz możemy przedstawić główne zależności za pomocą tego programu.<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-webflux</artifactId>  
 <scope>provided</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.immutables</groupId>  
 <artifactId>value</artifactId>  
 <version>2.7.1</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>com.fasterxml.jackson.datatype</groupId>  
 <artifactId>jackson-datatype-guava</artifactId>  
 <version>2.7.3</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-collections4</artifactId>  
 <version>4.3</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>  
 <dependency>  
 <groupId>io.projectreactor</groupId>  
 <artifactId>reactor-test</artifactId>  
 <version>3.2.10.RELEASE</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <version>5.2.0</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-params</artifactId>  
 <version>5.2.0</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.mockito</groupId>  
 <artifactId>mockito-core</artifactId>  
 <version>5.2.0</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.mockito</groupId>  
 <artifactId>mockito-junit-jupiter</artifactId>  
 <version>2.26.0</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>net.bytebuddy</groupId>  
 <artifactId>byte-buddy</artifactId>  
 <version>1.9.12</version>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

Do zbudowania modułu testowego użyjemy bibliotek: junit, mockito, bytebuddy, spring webflux, jackson, immutables.

Rozważmy junit.

JUnit jest frameworkiem testów jednostkowych dla języka programowania Java. JUnit był ważny w rozwoju rozwoju opartego na testach i jest jednym z rodziny frameworków do testowania jednostkowego, które są wspólnie znane jako xUnit, które pochodzą z SUnit.

JUnit jest połączony jako JAR w czasie kompilacji. Najnowsza wersja frameworka, JUnit 5, znajduje się w pakiecie org.junit.jupiter [10]. Poprzednie wersje JUnit 4 [10] i JUnit 3 znajdowały się odpowiednio w pakietach org.junit i junit.framework.

Każda klasa testowa JUnit ma zwykle kilka przypadków testowych. Te przypadki testowe podlegają cyklowi życia testu. Pełny cykl życia JUnit składa się z trzech głównych faz [15]:1) Konfiguracja.

W tej fazie przygotowywana jest infrastruktura testowa. Dostępne są dwa poziomy konfiguracji. Pierwszym typem konfiguracji jest konfiguracja na poziomie klasy, w której kosztowny obliczeniowo obiekt, taki jak połączenie z bazą danych, jest tworzony i ponownie wykorzystywany przy minimalnych efektach ubocznych. Konfiguracja na poziomie klasy jest implementowana przy użyciu adnotacji @BeforeAll. Drugi typ jest konfigurowany przed uruchomieniem każdego przypadku testowego, który wykorzystuje adnotację @BeforeEach [2].

2) Wykonanie testu.

Ta faza jest odpowiedzialna za uruchomienie testu i weryfikację wyniku. Wynik testu wskaże, czy test zakończył się sukcesem, czy porażką. Używana jest tutaj adnotacja @Test [2].

3) Sprzątanie.

Po wykonaniu wszystkich wykonań potestowych system może wymagać oczyszczenia. Podobnie jak w przypadku konfiguracji na poziomie klasy, istnieje odpowiedni poziom czyszczenia klasy. Adnotacja @AfterAll jest używana do obsługi czyszczenia na poziomie klasy. Adnotacja @AfterEach umożliwia czyszczenie po wykonaniu testu [2].

JUnit podąża za paradygmatem preferowania punktów rozszerzeń zamiast funkcji [7]. Zespół JUnit zdecydował się nie umieszczać wszystkich funkcji w rdzeniu JUnit, a zamiast tego postanowił zapewnić programistom rozszerzalny sposób rozwiązywania ich problemów [7]. W JUnit 4 zostało to osiągnięte przy użyciu API Runner i API Rule [11]. W JUnit 5 API rozszerzeń znajduje się w JUnit Jupiter Engine [12]. Zespół JUnit chce dać programistom możliwość podpięcia się do oddzielnych etapów cyklu życia testu [12]. W szczególności ma to miejsce, gdy silnik Jupiter jest w stanie wywołać wszystkie zarejestrowane rozszerzenia po osiągnięciu określonej fazy cyklu życia [12]. Deweloper może podłączyć się do pięciu głównych punktów rozszerzeń [12]:1. Wywołania zwrotne cyklu życia testu - pozwala to programiście na podłączenie się do określonych faz cyklu życia testu [13].

2. Post-processing instancji testowej - umożliwia programiście hakowanie po utworzeniu instancji testowej poprzez implementację interfejsu TestInstancePostProcessor [13].

3. Warunkowe wykonanie testu - umożliwia programiście wykonanie przypadku testowego tylko po spełnieniu określonych kryteriów [8].

4. Rozwiązywanie parametrów - umożliwia programiście rozwiązanie parametru po otrzymaniu go z metody testowej lub konstruktora.

5. Obsługa wyjątków - Przypadkiem użycia obsługi wyjątków jest zmiana zachowania testowego zamiast rzucania wyjątku [9].

Obiekt testowy JUnit jest obiektem Java. Metody testowe muszą być opatrzone adnotacją @Test. Jeśli wymaga tego sytuacja [10], możliwe jest również zdefiniowanie metody do wykonania przed (lub po) każdą (lub wszystkimi) metodami testowymi z adnotacjami @BeforeEach (lub @AfterEach) i @BeforeAll (lub @AfterAll) [4].

Weźmy pod uwagę mockito.

Mockito jest frameworkiem testowym open source dla Javy wydanym na licencji MIT [2,3]. Framework umożliwia tworzenie podwójnych obiektów testowych (obiektów mock) w zautomatyzowanych testach jednostkowych w celu rozwoju opartego na testach (TDD) lub rozwoju opartego na zachowaniu (BDD).

Mockito pozwala programistom zweryfikować zachowanie testowanego systemu (SUT) bez wcześniejszego ustalania oczekiwań [1]. Jednym z zarzutów wobec obiektów mock jest ścisłe powiązanie kodu testowego z testowanym systemem [6]. Mockito próbuje wyeliminować wzorzec expect-run-verify [5] poprzez usunięcie specyfikacji oczekiwań. Mockito zapewnia również pewne adnotacje w celu zmniejszenia ilości standardowego kodu [17].

Rozważmy ByteBuddy.ByteBuddy to biblioteka oprogramowania Java do manipulowania i tworzenia kodu bajtowego Java. Manipulacja kodem bajtowym może być wykonywana przed wykonaniem lub w trakcie działania aplikacji. Byte Buddy wykorzystuje bibliotekę ObjectWeb ASM do stosowania swoich manipulacji, ale zapewnia interfejs API wysokiego poziomu, który pozwala uniknąć wyraźnej specyfikacji kodu bajtowego Java, która jest wymagana przez ObjectWeb ASM. Zamiast tego Byte Buddy używa języka specyficznego dla domeny do wyrażania manipulacji kodem bajtowym na wysokim poziomie, gdzie czasowniki języka domeny opierają się na terminach języka programowania Java. Byte Buddy jest używana przez szeroką gamę komercyjnych i darmowych produktów na maszynie wirtualnej Java. Na przykład, biblioteka jest używana przez popularne biblioteki Mockito i Hibernate. Byte Buddy jest szczególnie popularny do tworzenia agentów Java, na przykład przez Instana APM Elastic APM, gdzie zmiany kodu są implementowane w zwykłym kodzie Java, który jest używany jako szablon do zmiany kodu klasy. Byte Buddy jest częścią OpenJDK Quality Outreach służącego do wczesnego przekazywania informacji zwrotnych na temat nadchodzących wydań Java. Biblioteka otrzymała nagrodę Duke's Choice Award w 2015 roku, a jej autor został wyróżniony nagrodą Oracle Groundbreaker Award w 2019 roku.

Rozważmy wiosenny webflux.

Spring WebFlux pozwala nam dekomponować logikę w deklaratywny sposób za pomocą Mono, Flux i ich bogatych zestawów operatorów. Co więcej, możemy mieć funkcjonalne punkty końcowe oprócz tych z adnotacją @Controller, chociaż możemy ich teraz używać również w Spring MVC.

Weźmy pod uwagę jackson.

W informatyce jackson jest wysokowydajnym procesorem JSON dla Javy. Jego twórcy wychwalają połączenie szybkich, poprawnych, lekkich i ergonomicznych atrybutów biblioteki. Jackson zapewnia wiele podejść do pracy z JSON, w tym wykorzystanie adnotacji wiążących na klasach POJO dla prostych przypadków użycia.

Weźmy pod uwagę Immutables.

Biblioteka składa się z adnotacji i procesorów adnotacji do generowania i pracy z serializowalnymi i konfigurowalnymi niezmiennymi obiektami.

Biblioteka generuje niezmienne obiekty z typów abstrakcyjnych: Interface, Class, Annotation.

Kluczem do osiągnięcia tego celu jest właściwe użycie adnotacji @Value.Immutable. Generuje ona niezmienną wersję typu z adnotacją i poprzedza jego nazwę słowem kluczowym Immutable.

1. **WYMAGANE DANE W POSTACI MODELI DO TESTOWANIA KOLEKCJI KART**
   1. **Models in the directory card/catalog.**
2. Interfaces:
3. Card.

@Immutable  
@JsonDeserialize(as = ImmutableCard.class)

public interface Card {  
  
 String name();  
 Set<Color> colorIdentity();  
 int convertedManaCost();  
}

1. CardCatalog.

public interface CardCatalog {  
  
 Flux<Card> getAllCards();  
 Flux<Card> matchCards(CardCriteria criteria);  
}

1. CardCriteria.

@Immutable

public interface CardCriteria {  
  
 default boolean exclusiveMatch() {  
 return true;  
 }  
  
 Optional<String> nameContains();  
 Optional<Set<Color>> colorIdentity();  
}

1. Enums.
2. Color.

public enum Color {  
 GREEN("G"), RED("R"), BLUE("U"), BLACK("B"), WHITE("W"), COLORLESS(null);  
  
 class lol {  
 }  
  
 private String abbreviation;  
  
 public static Color parse(String representation) {  
 return EnumSet.allOf(Color.class).stream()  
 .filter(color -> color.name().startsWith(representation) || color.matchesAbbreviation(representation))  
 .findFirst()  
 .orElseThrow(() -> new NoSuchElementException(format("No Color matching %s found.", representation)));  
 }  
  
 private boolean matchesAbbreviation(String representation) {  
 switch (this) {  
 case COLORLESS:  
 return false;  
 default:  
 return this.abbreviation.equals(representation);  
 }  
 }  
  
 Color(String abbreviation) {  
 this.abbreviation = abbreviation;  
 }  
}

1. Rarity.

public enum Rarity {  
 COMMON, UNCOMMON, RARE, MYTHIC\_RARE  
}

1. Type.

public enum Type {  
 *LAND*, *CREATURE*, *ENCHANTMENT*, *INSTANT*, *SORCERY*}

* 1. **Models in the directory card/client.**

Interfaces:

1. Page.

@Immutable  
public interface Page {  
  
 List<RawCard> cards();  
 Optional<Integer> nextPageNumber();  
 Integer lastPageNumber();  
}

1. RawCard.

@Immutable  
@JsonDeserialize(as = ImmutableRawCard.class)  
public interface RawCard {  
  
 String name();  
 String rarity();  
 double cmc();  
  
 Optional<String> text();  
 Optional<String> manaCost();  
 Set<String> colorIdentity();  
 Set<String> colors();  
 Set<String> types();  
}

1. RawCards.

@Immutable(builder = false)  
@JsonDeserialize(as = ImmutableRawCards.class)  
public interface RawCards {  
  
 @Parameter  
 List<RawCard> cards();  
}

1. CardsClient.

public interface CardsClient {  
  
 @Cacheable(value = "cardPages", cacheManager = "cardPagesCacheManager")  
 Mono<Page> getPage(int index);  
 Mono<Integer> getLastPageNumber();  
}

1. **COLLECTION OF TESTS**
   1. **Ordinary tests.**
      1. **RedundantTrieTest.**
2. GetRequiresMatch.

@Test  
 void get\_requiresExactMatch() {  
 final Trie<String> trie = Trie.*withKeyMapping*(Function.*identity*());  
 trie.add("abcd");  
 trie.search("bbcd", 2);

}

1. GetMultipleBranchesRequiresExactMatch.

@Test  
void get\_multipleBranches\_requiresExactMatch() {  
 final Trie<String> trie = Trie.*withKeyMapping*(Function.*identity*());  
 trie.add("abc");  
 trie.add("ax");

}

* + 1. **TestResourceGenerator.**

@Test  
 void saveOneThousandCardsAsJsonResource() {  
// final String json = cardSerializer.write(cardCatalog.getAllCards().take(1000).collect(Collectors.toList()).block());  
 }

* + 1. **MtgTestApplicationTests.**

@Test  
void contextLoads() {  
}

* 1. **Integration tests.**
     1. **MagicTheGatheringCardCatalogIntegrationTest.**

1. PrintCards.

@Test  
void printCards() {  
 cardCatalog.getAllCards().take(n).doOnEach(*out*::println).blockLast();  
}

1. CountCards.

@Test  
void countCards() {  
 *out*.println(  
 *format*("Counted %d Magic: The Gathering cards (taking n: %d)", cardCatalog.getAllCards().take(n).count().block(), n)  
 );  
}

1. PrintCardsWithGreenColorIdentity.

@Test  
void printCardsWithGreenColorIdentity() {  
 cardCatalog.matchCards(  
 ImmutableCardCriteria.*builder*()  
 .colorIdentity(Set.*of*(Color.*GREEN*))  
 .build()  
 ).take(n / 4).doOnEach(*out*::println).blockLast();  
}

1. PrintUniqueCard.

@Test  
void printUniqueCard() {  
 cardCatalog.matchCards(  
 ImmutableCardCriteria.*builder*()  
 .nameContains("Nekrataal")  
 .build()).take(1).doOnEach(System.*out*::println).blockLast();  
}

1. PrintCardsWithTheInName.

@Test  
void printCardsWithTheInName() {  
 cardCatalog.matchCards(  
 ImmutableCardCriteria.*builder*()  
 .nameContains("the")  
 .build()).take(n / 4).doOnEach(System.*out*::println).blockLast();  
}

* + 1. **MagicTheGatheringCardClientIntegrationTest.**

1. GetLastPageNumber.

@Test  
void getLastPageNumber() {  
 *assertThat*(cardsClient.getLastPageNumber().block()).isEqualTo(cardsClient.getPage(1).block().lastPageNumber());  
}

1. PrintCardsOnFirstPage.

@Test  
void printCardsOnFirstPage() {  
 *printCardsOnPage*(cardsClient.getPage(1));  
}

1. PrintCardsOnSeondPage.

@Test  
void printCardsOnSecondPage() {  
 *printCardsOnPage*(cardsClient.getPage(2));  
}

1. PrintCardsOnLastPage.

@Test  
void printCardsOnLastPage() {  
 *printCardsOnPage*(cardsClient.getPage(cardsClient.getLastPageNumber().block()));  
}

* + 1. **CardSearchTrieIntegrationTest.**

1. BuildAndSearch.

@Test  
void buildAndSearch() {  
 final Trie<Card> trie = Trie.*withKeyMapping*(Card::name);  
 cardCatalog.getAllCards().take(400)  
 .doOnNext(System.*out*::println)  
 .doOnNext(trie::add).blockLast();

}

1. GetMeNames.

@Test  
void getMeNames() {  
 System.*out*.println(  
 cardCatalog.getAllCards().take(400).map(Card::name).collect(Collectors.*toList*()).block().stream().collect(Collectors.*joining*("\\\",\\\""))  
 );  
}

* 1. **Unit tests.**
     1. **MagicTheGatheringCardCatalogUnitTest.**

1. GetAllCards for SinglePage.

@Test  
void getAllCards() {  
 *when*(cardsClient.getPage(1)).thenReturn(Mono.*just*(ImmutablePage.*builder*()  
 .nextPageNumber(Optional.*empty*())  
 .lastPageNumber(1)  
 .addCards(*RAW\_FLAMEWAVE\_INVOKER*, *RAW\_CANOPY\_SPIDER*)  
 .build())  
 );  
  
 StepVerifier.*create*(cardCatalog.getAllCards())  
 .expectNext(*MAPPED\_FLAMEWAVE\_INVOKER*)  
 .expectNext(*MAPPED\_CANOPY\_SPIDER*)  
 .verifyComplete();  
}

1. GetAllCards for MultiplePages.

@Test  
void getAllCards() {  
 StepVerifier.*create*(cardCatalog.getAllCards())  
 .expectNext(*MAPPED\_FLAMEWAVE\_INVOKER*)  
 .expectNext(*MAPPED\_CANOPY\_SPIDER*)  
 .expectComplete();  
}

1. MatchCardsByCardNameAllMatch for MultiplePages.

@Test  
void matchCards\_byCardName\_allMatch() {  
 StepVerifier.*create*(cardCatalog.matchCards(ImmutableCardCriteria.*builder*().nameContains("o").build()))  
 .expectNext(*MAPPED\_FLAMEWAVE\_INVOKER*)  
 .expectNext(*MAPPED\_CANOPY\_SPIDER*)  
 .expectComplete();  
}

1. MatchCardsByCardNameOneMatches for MultiplePages.

@Test  
void matchCards\_byCardName\_oneMatches() {  
 StepVerifier.*create*(cardCatalog.matchCards(ImmutableCardCriteria.*builder*().nameContains("flame").build()))  
 .expectNext(*MAPPED\_FLAMEWAVE\_INVOKER*)  
 .expectComplete();  
}

* + 1. **CardSearchTrieUnitTest.**

1. ExactMatchOnlyOneResult.

@Test  
void exactMatch\_onlyOneResult() {  
 final Card birds = *getCardByName*("Birds of Paradise");  
 *assertThat*(trie.search(birds.name(), 0)).isEqualTo(List.*of*(birds));  
}

1. DistanceOfOneTwoPrefixMatches.

@Test  
void distanceOfOne\_twoPrefixMatches() {  
 System.*out*.println(trie.search("Angel ", 3));

* + 1. **RedundantTrieUnitTest.**

1. DistanceOfZeroOneExactMatchOnePrefixMatch.

@Test  
void distanceOfZero\_oneExactMatch\_onePrefixMatch() {  
 *assertThat*(trie.search("Adam", 0)).containsExactlyInAnyOrder("Adam", "Adamology");  
}

1. DistanceOfOneOneApproximateMatchOneApproximatePrefixMatch.

@Test  
void distanceOfOne\_oneApproximateMatch\_oneApproximatePrefixMatch() {  
 *assertThat*(trie.search("Aaam", 1)).containsExactlyInAnyOrder("Adam", "Adamology");  
}

1. DistanceOfOneThreeApproximatePrefixMatches.

@Test  
void distanceOfOne\_threeApproximatePrefixMatches() {  
 *assertThat*(trie.search("Ad", 1)).containsExactlyInAnyOrder("Adam", "Ddom", "Adom", "Adelle", "Adamology");  
}

1. DistanceOfTwoTowoApproximateMatcgesOneApproximatePrefixMatch

@Test

void distanceOfTwo\_twoApproximateMatches\_oneApproximatePrefixMatch() {

assertThat(trie.search("Aaam", 2)).containsExactlyInAnyOrder("Adam", "Adom", "Adamology");

}

5) PrefixMatchIsExcludedWhenDistanceIsGreaterThanOne.

@Test

@Disabled("W pewnym momencie miałem bardziej rygorystyczne wymagania dotyczące odległości dla dopasowań prefiksów, ale testuję bardziej łagodną politykę.")

void prefixMatchIsExcludedWhenDistanceIsGreaterThanOne() {

assertThat(trie.search("Ddom", 2)).containsExactlyInAnyOrder("Adam", "Adom", "Ddom");

}

**WNIOSKI**

W tej pracy budujemy program do testowania api: https://api.magicthegathering.io/v1/cards, który zawiera wiele różnych testów jednostkowych, integracyjnych i zwykłych. Testy te działają niezależnie od siebie i dają użytkownikom pewne wyniki dotyczące określonych kontroli/błędów.

**LISTA WYKORZYSTANYCH ŹRÓDEŁ**

1. ["Features and Motivations"](https://code.google.com/p/mockito/wiki/FeaturesAndMotivations). Retrieved 2010-12-29.
2. ["Mockito in six easy examples"](http://gojko.net/2009/10/23/mockito-in-six-easy-examples/). Retrieved 2012-10-05, 2009.
3. ["What's the best mock framework for Java?"](https://stackoverflow.com/questions/22697/whats-the-best-mock-framework-for-java). Retrieved 2010-12-29.
4. [*"Writing Tests"*](https://junit.org/junit5/docs/current/user-guide/#writing-tests). junit.org*. Retrieved 2021-02-04*.
5. Faber, Szczepan. ["Death Wish"](https://web.archive.org/web/2010/http:/monkeyisland.pl/2008/02/01/deathwish/). Retrieved 2010-12-29.
6. Fowler, Martin. ["Mocks Aren't Stubs"](http://martinfowler.com/articles/mocksArentStubs.html#CouplingTestsToImplementations). Retrieved 2010-12-29.
7. Gulati & Sharma, p. 121, Chapter §7 JUnit 5 Extension Model, 2017
8. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), p. 127, Chapter §7 JUnit 5 Extension Model - Conditional Test Execution.
9. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), p. 129, Chapter §7 JUnit 5 Extension Model - Exception Handling.
10. Gulati & Sharma, p. 144, §Chapter 8 Dynamic Tests and Migration from Junit 4, 2017.
11. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 121–122, Chapter §7 JUnit 4 Extension Model, 2017
12. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 122–124, Chapter §7 JUnit 5 Extension Model - JUnit 5 Extension Model.
13. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 124–126, Chapter §7 JUnit 5 Extension Model - Test Life Cycle Callbacks.
14. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 126–127, Chapter §7 JUnit 5 Extension Model - Test Instance Post-Processing.
15. Gulati & Sharma, pp. 37–40, Chapter §2 JUnit LifeCycle API, 2017.
16. Kaczanowski, Tomek. ["Mockito - Open Source Java Mocking Framework"](http://www.methodsandtools.com/tools/mockito.php). Retrieved 2013-09-17.
17. [Kent Beck](https://en.wikipedia.org/wiki/Kent_Beck). ["Expensive Setup Smell"](http://c2.com/cgi/wiki?ExpensiveSetUpSmell). C2 Wiki. Retrieved 2011-11-28.