AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Damian Łączak

kierunek studiów: **Informatyka Stosowana**

Aplikacja internetowa do nauki języka

Opiekun: **dr hab. inż Tomasz Bołd**

Kraków, styczeń, 2018

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

Spis treści

Todo list

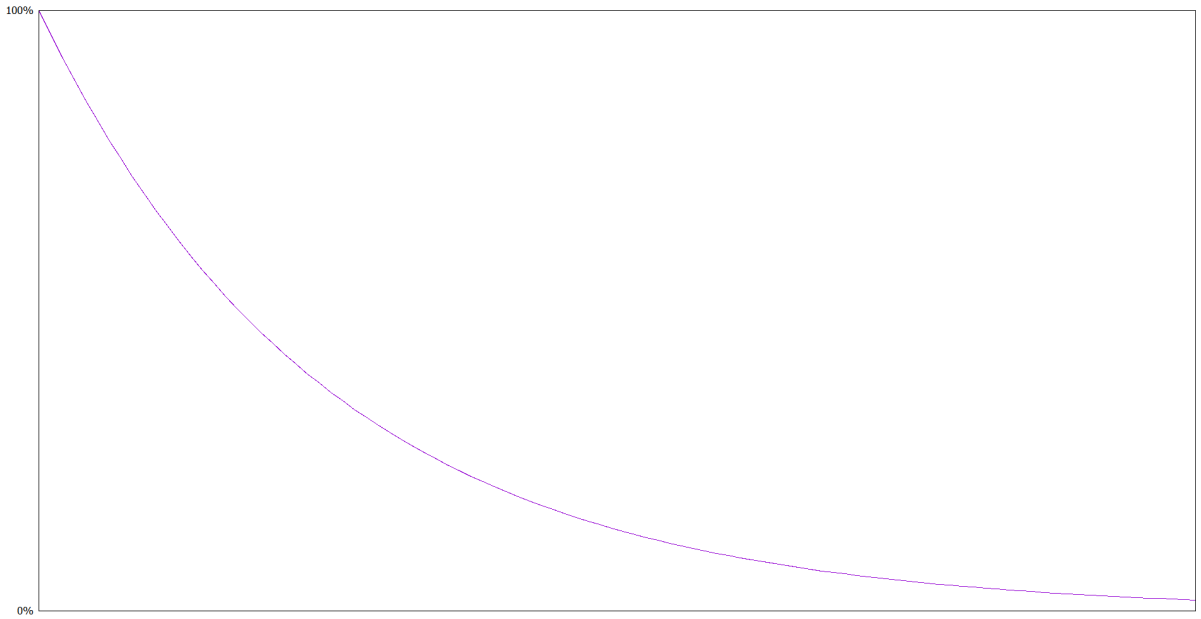
1. Wstęp

TODO WSTEP

2. Przyswajanie wiedzy

Jeśli opisuje coś na podstawie strony internetowej pana Doktora to czy to jest plagiat?

Hermann Ebbinghaus[1] (1850-1909) był Niemieckim psychologiem, który jako jeden z pierwszych zajmował się tematyką eksperymentalnej psychologii związanej z przyswajaniem wiedzy. Jednym z jego pierwszych eksperymentów było stworzenie testu, podczas którego uczył się zestawu 20 sylab, które były bezsensowne ze względu na fakt, iż w jego języku nie występowały żadne słowa, które ich używały. Eksperyment ten pozwolił mu skonstruować pierwszą na świecie krzywą zapominania. Miała ona charakter eksponencjalny. Możemy z niej wywnioskować, iż podczas początkowego okresu zapominania tracimy najwięcej zapamiętanych informacji.



Rys. 2.1. Eksponencjalna krzywa zapominania.

Trzeba także zwrócić uwagę na fakt, iż w rzeczywistości[2] nasz mózg nie jest w stanie przyswoić danych informacji w 100% po zakończeniu nauki.

2.1. Interwały potwórzeń

W celu jak najlepszego zapamiętania wyuczonych informacji należy, poza odpowiednim programem nauczania, zwrócić uwagę na fakt, w jakich interwałach czasowych powtarzamy przerabiany materiał[2]. Interwał musi być wystarczająco krótki, aby zapobiec zapominaniu i wystarczająco długi, aby zbyt często nie przyswajać powtarzanego materiału.

Można tu jeszcze coś napisać

3. .NET

.NET jest platformą programistyczną umożliwiającą pisanie nowoczesnych aplikacji w językach wysokiego poziomu, do których zalicza się m.in C#, VB oraz F#. Platforma ta wyróżnia się tym iż:

- Pozwala na użycie wielu języków programowania podczas pisania naszych programów.
- Ma zaimplementowane mechanizmy do obsługi operacji asynchronicznych i współbieżnych.
- Można ją stosować na różnych platformach, które posiadają środowisko wykonywalne .NET.

Wszystkie języki używane w platformie .NET kompilowane są do Wspólnego Języka Pośredniego (po ang. *Common Intermediate Language*), który następnie jest tłumaczony na kod bajtowy i wykonywany za pomocą środowiska wykonywalnego danej implementacji .NET.

```
.assembly HelloWorld
.class auto ansi HelloWorldApp
{
     .method public hidebysig static void Main() cil managed
     {
          .entrypoint
          .maxstack 1
          ldstr "Hello world."
          call void [mscorlib]System.Console::WriteLine(string)
          ret
     }
}
```

Listing 3.1. Przykładowy kod aplikacji "Hello World"w języku CIL

3.1. Implementacje .NET

Każda aplikacja .NET jest uruchamiana na jednej z implementacji .NET.
Od roku 2016 wprowadzono .NET Standard - wspólny zestaw API, które każda z implementacji musi posiadać. Pozwala to na pisanie i używanie bibliotek programistycznych w różnych środowiskach .NET.

Istnieją aktualnie 4 główne implementacje .NET:

3.1.1. .NET Core

Został napisany z myślą o tworzeniu aplikacji cross-platformowych, które mogą zostać uruchomione na serwerach, jak i środowiskach chmurowych. Potrafi działać na platformie Windows, macOS oraz Linux. Jest to pierwsza implementacja .NET, która została zaprojektowana przez Microsoft z myślą o wieloplatformowości.

3.1.2. .NET Framework

Jest to pierwsza, oryginalna implementacja .NET, która istnieje od roku 2002. Składa się ze środowiska uruchomieniowego Common Language Runtime (CLR) oraz biblioteki standardowej zwanej jako Framework Class Library (FCL). CLR zapewnia aplikacjom wirtualną maszynę, na której wykonywany kod bajtowy skompilowany z języka CIL. Ta implementacja jest używana w tej pracy inżynierskiej.

3.1.3. Mono

Darmowy projekt open-source prowadzony przez firmę Xamarin. Powodem stworzenia tego produktu była możliwość uruchamiania aplikacji napisanych w językach .NET na wielu platformach, jak i dostarczenie użytkownikom Linuxa narzędzi pozwalających na aplikacji w rodzinie języków .NET.

3.1.4. Universal Windows Platform (*UWP*)

Implementacja, która umożliwia tworzenie aplikacji dla wszystkich platform używających Windows 10, Xboxa, niektórych urządzeń stworzonych przez Microsoft i dostosowanych urządzeń IoT.

4. ASP.NET MVC

ASP.NET MVC jest frameworkiem do budowania aplikacji internetowych w oparciu o wzorzec architektoniczny Model-View-Controller (MVC). Wykorzystuje implementacje .NET Framework do uruchamiania skompilowanego kodu źródłowego.

4.1. Model-Widok-Kontroler

Większość dzisiejszych systemów komputerowych działa na zasadzie wyświetlania danych, które aktualnie znajdują się w bazie danych i ewentualnie ich modyfikacji. W celu ujednolicenia tych systemów stosowane jest wzorzec Model-Widok-Kontroler(ang. Model-View-Controller), który rozdziela logikę aplikacji na 3 główne segmenty:

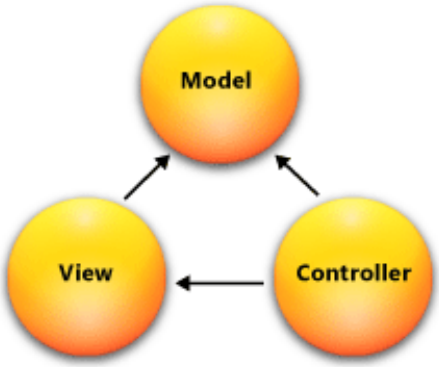
1. Model - Służy do pobierania, przechowywania i zamiany danych.

2. Kontroler - przetwarza zapytania użytkownika, które przekazuje do modelu, który jest następnie wyświetlany jako widok.

3. Widok - Służy do wyświetlania informacji

Cała aplikacja została skonstruowana zgodnie z tym wzorcem projektowym. ASP.NET MVC posiada wiele narzędzi, które ułatwiają zastosowanie tego wzorca. Dostarczana jest klasa bazowa **Controller**, która posiada wszystkie podstawowe metody do wyświetlania odpowiednich treści danych i zarządzania zapytaniami.

Wzorzec MVC niesie za sobą korzyści związane z warstwą wyświetlania danych. Jest to warstwa aplikacji, w której bardzo często dochodzi do zmian. Dzięki odseparowaniu danych od widoku jesteśmy w stanie tworzyć, jak i zmieniać widoki, bez wpływu na kod biznesowy aplikacji.

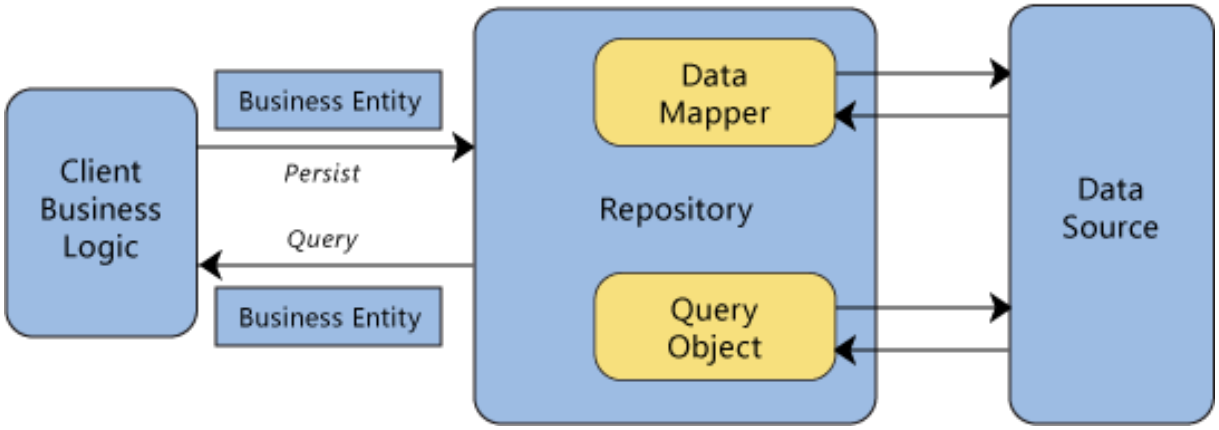


Rys. 4.1. Podać źródło

4.2. Wzorzec Repozytorium[3]

Wewnątrz aplikacji będziemy używać bazy danych Microsoft SQL. Do komunikacji z bazą bardzo często jest tworzony kod, który zwraca podobne dane. W celu zmniejszenia redundancji kodu, jak i odseparowania zależności i odpowiedzialności wykorzystałem wzorzec Repozytorium (z ang. Repository Pattern).

Wzorzec ten wykorzystuje obiekty, zwane repozytoriami, których zadaniem jest pobieranie i modyfikowanie danych po stronie serwera SQL. Nie zawierają żadnej logiki biznesowej i są niezwiązane z resztą kodu danej aplikacji.

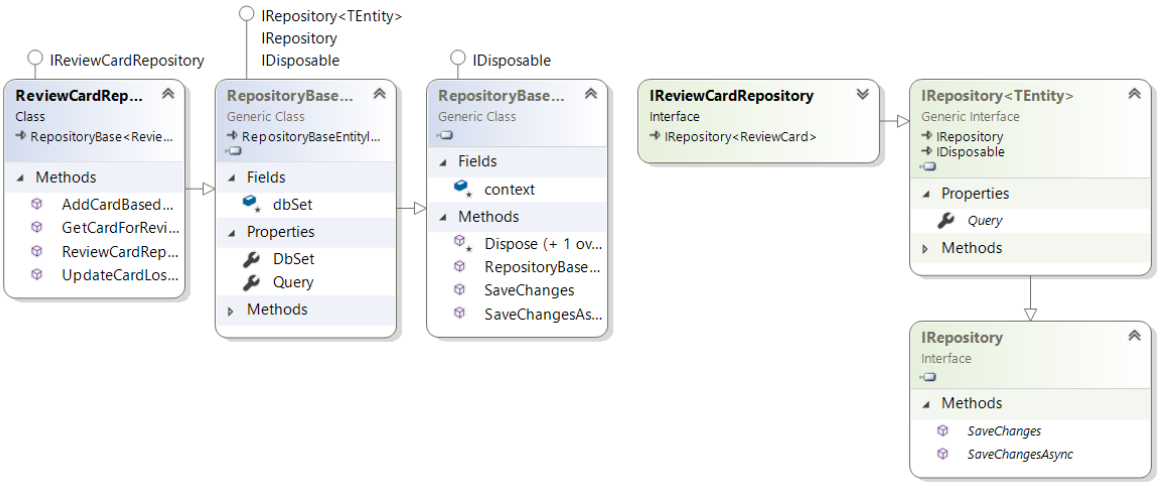


Rys. 4.2. Wzorzec repozytorium. Źródło: msdn.microsoft.com

źródło podmień?

Wewnątrz projektu cały wzorzec repozytorium został oparty na interfejsach **IRepository** i **IRepository<T>** (który implementuje **IRepository**). W założeniu te interfejsy definiują operacje, które można zrealizować na danym obiekcie klasy 'T'. Informacje o tym jak te informacje są realizowane, jak i to, iż są one realizowane za pomocą bazy danych są przechowywane w klasie **RepositoryBase<T>**. Wszystkie kolejne stworzone repozytoria dziedziczą po **RepositoryBase<T>** a ich interfejsy implementują **IRepository<T>**. Takie działanie pozwala

nam na separację reprezentacji persystentnej (baza danych) i reprezentacji bieżącej (repozytoria). Wystarczy, iż zmienimy **RepositoryBase<T>** na jakąkolwiek inną klasę implementującą **IRepository<T>** i wykorzystamy ją w naszych repozytoriach. Dzięki temu możemy zastąpić Microsoft SQL inną implementacją.



Rys. 4.3. Repozytorium ReviewCardRepository wraz z implementowanymi interfejsami i odziedziczonymi klasami.

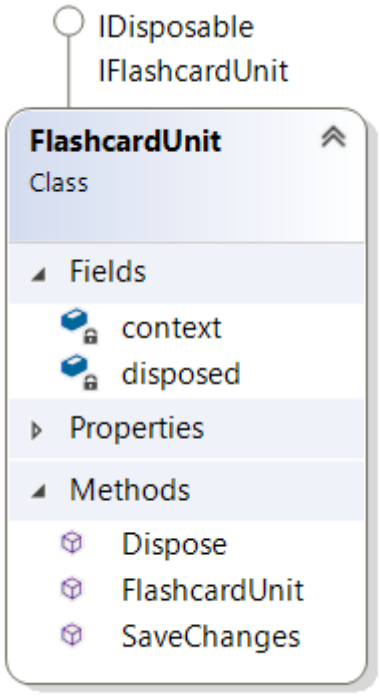
4.3. Jednostka Pracy[3]

Wzorzec Jednostki Pracy (z ang. Unit of Work) ma na celu uporządkowanie pracy z repozytoriami za pomocą umieszczenia ich wszystkich w jednej klasie. Dodatkowo wszystkie repozytoria współdzielą kontekst dostępu do bazy danych. Zapisanie danych odbywa się poprzez wywołanie metody **SaveChanges** wewnątrz Jednostki Pracy (SaveChanges jest konwencją wewnątrz mojego projektu).

4.4. Mapowanie obiektowo-relacyjne

Wewnątrz mojego projektu wykorzystałem narzędzie Entity Framework, które pozwala na korzystanie z mapowania obiektowo-relacyjnego. Dzięki tej technice mogę uprzednio zdefiniowane dane tabelaryczne odwzorować za pomocą klas, które zostaną stworzone na podstawie definicji tabeli i relacji między nimi. Powstałe klasy są używane wewnątrz kolekcji, implementujących interfejs **IQueryable<T>**, na których możemy wykonywać zapytania z pomocą **LINQ**a, które następnie zostaną przetłumaczone na zapytania SQL.

Takowe rozwiązanie ma wiele zalet :



Rys. 4.4. Jednostka pracy wykorzystywana w projekcie.

1. Dane z danej tabeli będziemy mogli otrzymać zawsze w tym samym formacie. Unika się dzięki temu sytuacji gdzie dana tabela ma wiele odpowiadających jej klas w kodzie, które służą jedynie do dostępu do danych.

2. W przypadku zmiany typu w tabeli typ ten możemy bardzo prosto zmienić w kodzie.

3. Zmniejsza to zakres potrzebnych umiejętności w celu używania danych występujących w bazie danych. Dany programista nie musi znać języka SQL, aby w sposób bezproblemowy pobrać interesujące go dane, nawet gdy tworzy bardzo skomplikowane zapytania.

4. Łatwiejsza nawigacja po zależnościach między tabelami. Na danym typie możemy na przykład wykorzystać operacje **Find All References**, która wskaże nam użycia danego typu w naszym kodzie. W przypadku kodu SQL nie mielibyśmy takiej możliwości.

Można napisać coś o lazy loading tego rozwiązania

4.5. Wstrzykiwanie zależności

Wstrzykiwanie zależności (

Informacja o tym, iż dana klasa ma zamiar używać instancji innej klasy może być zawarta poprzez odpowiednie wywołanie danej funkcji, bądź poprzez użycie listy interfejsów w konstruktorze, która zostanie wypełniona odpowiednimi instancjami.

W przypadku mojej aplikacji użyłem do tego biblioteki Ninject[4]. Jest to bardzo popularna biblioteka realizująca wzorzec wstrzykiwania zależności. Dzięki niej możemy zdefiniować listę potrzebnych interfejsów/klas jako parametry konstruktora naszego kontrolera, a te utworzą się automatycznie podczas żądania użytkownika. Jeśli tworzone klasy także będą wymagały do swojego istnienia pewnych interfejsów bądź klas i zostanie to uściślone w ich konstruktorze to także dla nich zostaną dostarczone odpowiednie instancje.

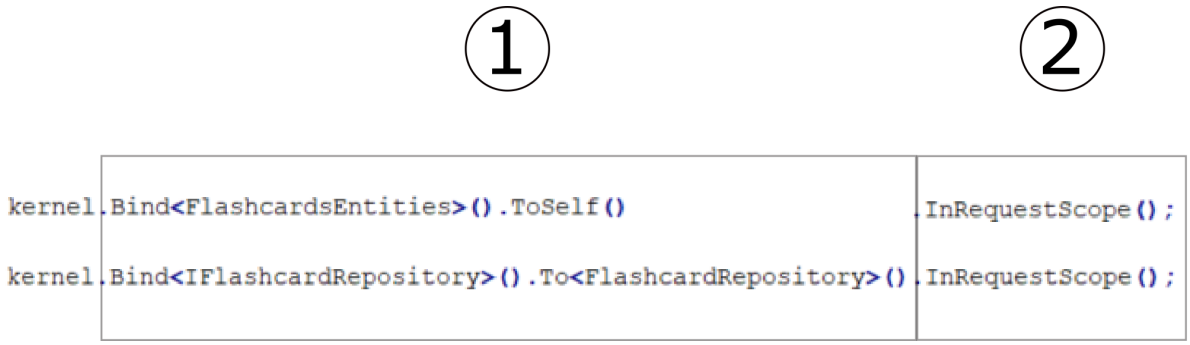
Jednakże sam interfejs nie daje żadnej informacji o tym, jaką klasę należy zinstancjować, ponieważ może być wykorzystywany przez wiele klas pochodnych. W tym celu należy poinformować Ninject o tym, jakie klasy mają zostać zinstancjonowane poprzez zbindowanie interfejsów do odpowiednich klas. To działanie jest przedstawione na poniższym listingu:
Nie zawsze chodzi tylko o interfejsy - uściślić

```
public static void RegisterServices(IKernel kernel)
{
kernel.Bind<FlashcardsEntities>().ToSelf().InRequestScope();
kernel.Bind<IFlashcardRepository>().To<FlashcardRepository>().InRequestScope();
}
```

Listing 4.1. Przykładowy kod bindowania dla biblioteki Ninject

Jak widzimy proces pojedynczego bindowania możemy podzielić na 2 części:

1. Informacja o tym co powinno zostać stworzone przez Ninject gdy jest proszone o daną klasę
2. Informacja o zasięgu współużywania danej instancji.



Rys. 4.5. Proces bindowania z wyszczególnionymi częściami składowymi

Wewnątrz pierwszej części określamy jakim typem ma zostać zastąpiony dany typ w procesie wstrzykiwania. Na przykładzie widzimy, że typ może źastąpićśam siebie bądź może zostać użyty inny typ, który będzie jego instancją. Należy pamiętać, iż ten drugi typ musi implementować, dziedziczyć bądź być tym samym typem co pierwszy.

Ninject dodatkowo potrafi zbindować dany typ do **metody** lub klasy implementującej interfejs **Provider<T>**. Taka metoda, badź klasa musi zwrócić w wyniku swojego działania pełnoprawny obiekt danego typu.

Ninject w swoim domyślnym zachowaniu za każdym razem jak poprosimy go obiekt danej klasy to tworzy on następny obiekt danej klasy. Jednakże często dochodzi do sytuacji, gdzie w danym segmencie kodu chcielibyśmy, aby obiekt danej klasy był współdzielony między wszystkimi instancjami wytworzonymi przez Ninject. Wtedy za każdym razem jak poprosimy obiekt klasy A to zawsze w ramach danego zasięgu dostaniemy ten sam obiekt i nie zostanie on utworzony ponownie. Jest to szczególnie przydatne w środowisku serwerowym, gdy chcemy, aby dla każdego zapytania klienta wszystkie repozytoria i serwisy były zawsze takie same. Nie tylko nie ponosimy kosztów utworzenia danej instancji, lecz także współdzielone repozytoria mogą szybciej zwracać informacje odnośnie danych po stronie serwera SQL, gdy zobaczą, iż mają te dane w swoim cache'u.

5. Microsoft SQL Server

Microsoft SQL Server[5] jest aplikacją, umożliwiającą zarządzanie serwerami baz danych. Został stworzony przez firmę Microsoft. Jego pierwsza edycja ukazała się w roku 1989.

Do tworzenia zapytań wykorzystano tutaj język Transact-SQL (T-SQL). Jest to rozszerzenie języka SQL (Structured Query Language), które między innymi wprowadza:

1. Lokalne zmienne.
2. Blok **Try Catch**, wspierający obsługę wyjątków.
3. Dodatkowe funkcjonalności związane z obsługą tekstu, kalendarza i matematyki.
4. Możliwość tworzenia pętli oraz instrukcji warunkowych.
5. Procedury, funkcje składowane i wyzwalacze.

W celu integracji serwera Microsoft SQL z projektem napisanym w języku C# należy pobrać paczkę NuGet[1] o nazwie **System.Data.SqlClient**.

[1]NuGet jest menadżerem paczek w środowisku .NET.[6]

6. Testy

Aplikacje tworzone w dzisiejszych czasach wymagają dostarczenia zestawu testów, których zadaniem jest sprawdzenie poprawności działania aplikacji. Z tego też powodu napisany został zestaw testów, które sprawdzają, czy poszczególne komponenty aplikacji działają poprawnie.

6.1. Testy Jednostkowe

Testy jednostkowe mają za zadanie sprawdzić poprawność działania pojedynczego modułu. Wszelkie odwołania do innych komponentów zostają zastąpione przez atrapy obiektów (po ang. mock objects), które symulują działanie ich prawdziwych odpowiedników. Wewnątrz projektu do operacji związanych z atrapami wykorzystany został framework Moq, który zawiera także dodatkowe funkcjonalności pozwalające ocenić czy dany test przebiegł poprawnie. Testy zostały ułożone zgodnie ze wzorcem AAA[7] - Aranżacja (po ang. Arrange) Akcja (po ang. Act) Asercja (po ang. assert). Dzięki temu testy są konsystentne i czytelne, przez co nie potrzeba dużej ilości czasu, aby zaznajomić się z nimi.

```
public void StopLastTraining_assert_tests()
{
        //Aranżacja - stworzenie wewnętrznej atrapy obiektu dla atrapy serwisu
        mockTraining();
        //Akcja
        trainingReviewService.StopLastTraining();
        //Asercja - sprawdzenie poprawności wykonania
        trainingRepository.Verify(x => x.Remove(It.IsAny<long>()), Times.Once);
        unit.Verify(x => x.SaveChanges(), Times.Once); //wykorzystanie ↩
            funkcjonalności frameworku Moq w celu sprawdzenia czy dana metoda ↩
            została wywołana.
        Assert.AreEqual(null, sessionService.Object.UserInfo.TrainingInfo);
}
```

Listing 6.1. Przykładowy test jednostkowy wykorzystujący wzorzec AAA

6.2. Testy Integracyjne

Testy integracyjne mają za zadanie wykryć potencjalne błędy występujące pomiędzy modułami.

Takowe testy nie sprawdzają

```
[TestMethod]
public void ExistTest()
{
        using (var temp = new WindowsTempFile())
                Assert.IsTrue(File.Exists(temp.Path));
}}
```

Listing 6.2. Test integracyjny wykorzystany w projekcie.

7. Kaskadowe Arkusze Styli

Kaskadowe Arkusze Styli (po ang. Cascading Style Sheets, w skrócie CSS)[**CSSDoc**] wykorzystywane są w celu opisu sposobu wyświetlania elementów stron WWW. Pozwalają zdefiniować między innymi takie właściwości elementów jak: kolory, czcionki, położenie. Dzięki oddzieleniu danych od sposobu ich wyświetlania możliwe jest wykorzystanie tych samych arkuszów styli na wielu stronach. W celu określenia wyglądu danego elementu należy zdefiniować odpowiednią regułę, na którą składa się :

1. Selektor, który określa dla jakich elementów przeznaczona jest dana reguła.

2. Zestaw właściwości wraz z przypisanymi do nich wartościami

```
p //seletor p
{
//Właściwość color do której przypisano wartość orange
    color: orange;
}
```

Listing 7.1. Przykład prostego selektora, który sprawi, iż wszystkie paragrafy będą miały domyślnie kolor pomarańczowy.

7.1. Syntaktycznie Zarąbiste Arkusze Styli

W dzisiejszym świecie nikt obeznany z technologią tworzenia stron internetowych nie używa już czystego języka CSS do tworzenia arkuszy styli. Wynika to z faktu występowania zbyt dużej redundancji kodu, która jest związana z tym, iż niektóre fragmenty strony są formatowane zawsze w ten sam sposób (np. motyw kolorystyczny dla większości elementów jest ten sam).

Jest to spowodowane brakiem zmiennych, szablonów, oraz innych konstrukcji, które pomogłoby w ograniczeniu powyższego problemu.

Z tego też powodu powstały rozwiązania niedogodności występujących w języku CSS. Jednym z nich jest język SASS - Syntactically Awesome Style Sheets (z ang. Syntaktycznie Zarąbiste Arkusze Styli)[8]. Jest on kompatybilny ze wszystkimi wersjami CSS, przez co każdy kod napisany w CSS jest z nim zgodny. Oferuje bardzo dużo funkcjonalności, między innymi:

1. Zmienne, które pozwalają wielokrotnie używać danych wartości
2. Możliwość zagnieżdżania reguł w innych regułach.
3. Możliwość załączania plików pozwalająca na podział arkuszy styli.
4. Mixin - szablony reguł, które można ponownie wykorzystywać.
5. Dziedziczenie reguł za pomocą słówka kluczowego **@extend**
6. Za pomocą operatorów +, -, *, /, % można wykonywać operacjach na liczbach.

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}

.box { @include border-radius(10px); }
```

Listing 7.2. Przykładowy kod SCSS wykorzystujący mixin.

8. Scala

Scala[9] jest językiem programowania ogólnego zastosowania, który został wprowadzony na rynek przez Laboratorium "École Polytechnique Fédérale de Lausanne". Jest to język kompilowalny bezpośrednio do kodu bajtowego Javy, przez co programy w nim napisane z łatwością uruchamiają się w środowisku wykonywalnym maszyny wirtualnej Javy. Scala jest językiem wielo-paradygmatowym[10]. Korzysta z dobrodziejstw programowania funkcjonalnego i obiektowego.

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

Listing 8.1. Hello world napisany w języku Scala.

8.1. Javascript

Javascript jest głównym językiem programowania wykorzystywanym na stronach internetowych. Może być on interpretowany bądź kompilowany metodą Just In Time, która kompiluje kod tuż przed jego wykonaniem. Jest to dynamiczny język wieloparadygmatowy bazujący na prototypach. Wspiera programowanie obiektowe, imperatywne i deklaratywne.

Javascript jako język posiada wiele problemów, które mogą doprowadzić do wystąpienia błędów, między innymi takich jak:

1. Niejawne rzutowania, które objawiają się przy użyciu operatora ==. Są przyczyną wielu trudnych do odkrycia błędów

2. Automatyczne wstawianie średników, co może spowodować, iż program działa inaczej niż zamierzał to programista. Czasami potrafi to uratować program napisany przez programistę, lecz często prowadzi do dziwnych i niezrozumiałych błędów, takich jak w przypadku poniższego listingu: 8.2

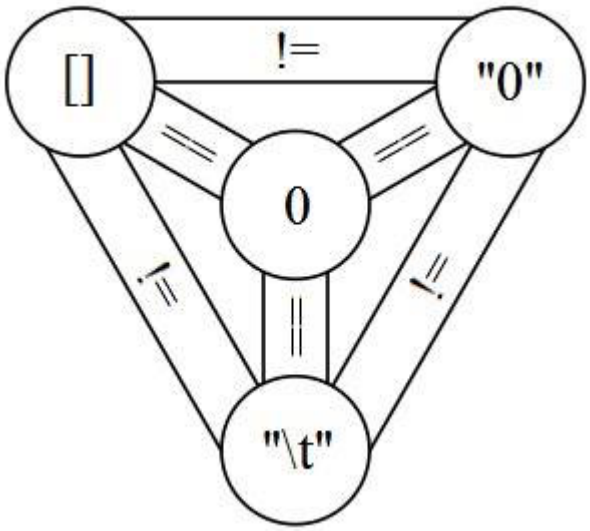
3. Niezrozumiałe zachowana różnych funkcji. Metoda sortująca będzie domyślnie sortowała tablice liczb w porządku alfabetycznym nie zwracając uwagi na fakt, iż w tablicy nie występuje ani jeden ciąg znakowy.

4. Zmienne, którym możemy w każdej chwili zmienić typ przechowywanej wartości. Z tego też powodu bardzo często powstają prozaiczne błędy, gdzie programista oczekując liczby w danym miejscu otrzymał ciąg znakowy co skutkuje błędem w dalszym kodzie programu.

```
function foo() {
    return // Tutaj zostanie wstawiony średnik.
        {
            bar : "test"
        };
}
```

Listing 8.2. Przykład niepoprawnego kodu Javascript wynikłego z automatycznego wstawienia średnika

Z tego też powodu powstały języki, które starają się być lepsze i przyjaźniejsze dla programisty niż Javascript. Jest to między innymi: Typescript, Coffescript, Dart czy Scala.js. Takie języki programowania nie są wykonywane/kompilowane bezpośrednio w przeglądarce, lecz wpierw są kompilowane do kodu Javascript. Spowodowane jest to używaniem przez przeglądarki jedynie języka Javascript do wykonywania kodu[1].



Rys. 8.1. Przykład dziwnego zachowania języka Javascript. Źródło: devrant.com

8.2. Scala w wersji JS

Scala.js jest językiem kompilowalnym do Javascripta, który umożliwia pozbycie się niedogodności związanych z używaniem języka Javascript. [11].

1. Silne typowanie, dzięki któremu pisany kod nie zawiera bardzo prostych błędów.

[1]Wyjątkiem jest tutaj przeglądarka Dartium, która potrafi posiada maszynę wirtualną języka Dart[**Dartium**]

2. Podczas programowania nie trzeba się skupiać na dziwnych i frustrujących aspektach języka Javascript.

8.2.1. sbt

sbt (Simple Build Tool) jest narzędziem o otwartym kodzie źródłowym, pozwalającym na zarządzanie procesem budowania aplikacji napisanej w języku Scala lub Java. Jest to narzędzie bardzo rozbudowane, które m.in. umożliwia:

1. Współpracę z wieloma narzędziami testującymi dla scali.
2. Inkrementacyjne kompilowanie i testowanie.
3. Zarządzanie zależnościami przy pomocy menadżera paczek Ivy.[**ApacheIvy**]
4. Ciągłą wdrażanie, kompilowanie i testowanie kodu.

2cm

W celu wykorzystania naszej aplikacji należy użyć polecenia **sbt new sbt/scala-seed.g8**, które spowoduje utworzenie minimalnego projektu Scali, który możemy dostosować do własnych potrzeb. Z tego też powodu musimy wykonać parę istotnych zmian w projekcie, aby był on kompilowalny do języka Javascript za pomocą Scali.js.

1. Należy dodać plik ./project/plugins.sbt[2], wewnątrz którego znajdzie się instrukcja **addSbtPlugin(órg.scala-js"% śbt-scalajs"% "0.6.20")**, która informuje o tym, iż chcemy użyć Scali.JS
2. W pliku ./project/build.properties musimy określić wersje używanego przez nas narzędzia sbt poprzez dodanie bądź zmodyfikowanie linijki:

```
sbt.version=0.13.16
```

3. Następnie musimy zmodyfikować plik build.sbt znajdujący się w głównym folderze projektu. Przykładowy plik, który jest używany w projekcie, znajduje się na listingu 8.4.

```
enablePlugins(ScalaJSPlugin)
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1"
libraryDependencies += "be.doeraene" %%% "scalajs-jquery" % "0.9.1"

name := "Flashcards"
scalaVersion := "2.12.2"

// Informacja o tym, iż chcemy aby nasz kod zawsze miał uruchamianą metodę main ↩
    po wczytaniu witryny.
```

[2]./ jest katalogiem głównym projektu w tym przypadku.

```
scalaJSUseMainModuleInitializer := true

skip in packageJSDependencies := false
jsDependencies += "org.webjars" % "jquery" % "2.1.4" / "2.1.4/jquery.js"
jsDependencies += "org.webjars.bower" % "jsrender" % "1.0.0-rc.70" / ↩
    "1.0.0-rc.70/jsrender.js"
```

Listing 8.3. Plik .sbt, który jest wykorzystywany w projekcie.

8.2.2. Proces budowania projektu

Projekt kompilacji projektu jest podzielony na 3 etapy[12]:

1. Początkową kompilację
2. Szybką optymalizację
3. Pełną optymalizację (Opcjonalny)

8.2.2.1. Początkowa kompilacja

W trakcie kompilacji pliki .scala są kompilowane do plików .class i .sjsir. Pliki .class nie biorą udziału w tworzeniu kodu javascript. Ich zadaniem jest współpraca z innymi narzędziami, które być może będą ich używać. Przykładem takiego narzędzia może być **IntelliJ** lub **Eclipse**, które plików .class używają w celu wspomagania programisty w trakcie pisania kodu. Pliki .sjsir (Nazwa rozszerzenia jest skrótem od ŠcalaJS Intermediate Representation")[13] zawierają kod przejściowy między Scalą a Javascriptem. Większość konstrukcji została zastąpiona przez ekwiwalenty z języka Javascript. Gdybyśmy połączyli wszystkie pliki .sjsir, wyprodukowane przez sbt to ich wynikiem byłby plik większy niż 20 MB. Wynika to z faktu, iż w tym pliku nadal znajduje się wiele niepotrzebnych bibliotek i konstrukcji. Jak na przykład **cała** biblioteka standardowa Scali.

```
module class Ltutorial_webapp_TutorialApp$ extends O {
  def main__AT__V(args: T[]) {
    this.appendPar__Lorg_scalajs_dom_raw_Node__T__V
    (mod:Lorg_scalajs_dom_package$.document__Lorg_scalajs_dom_raw_HTMLDocument()["body"], ↩
        "Hello World")
  }
  def appendPar__Lorg_scalajs_dom_raw_Node__T__V(targetNode: any, text: T) {
    val parNode: any = ↩
        mod:Lorg_scalajs_dom_package$.document__Lorg_scalajs_dom_raw_HTMLDocument()
    ["createElement"]("p");
    val textNode: any = ↩
        mod:Lorg_scalajs_dom_package$.document__Lorg_scalajs_dom_raw_HTMLDocument()
    ["createTextNode"](text);
    parNode["appendChild"](textNode);
```

```
    targetNode["appendChild"](parNode)
  }
  def init___() {
    this.O::init___();
    mod:Ltutorial_webapp_TutorialApp$<-this
  }
}
```

Listing 8.4. Przykładowy plik .sjsir dla projektu wyświetlającego HelloWorld na ekranie.

8.2.2.2. Szybka optymalizacja

W celu optymalizacji poprzedniego kroku stosuje się szybką optymalizację kodu .sjsir, która jako rezultat swej pracy stworzy kod Javascript. W tym celu należy użyć optymalizatora FastOptJS poprzez wpisanie komendy **FastOptJS** w sbt. Optymalizacja ma na celu:

1. Wyeliminowanie fragmentów kodu, które są nieużywane. Na przykład kod biblioteki standardowej, który nie zostanie wywołany.

2. Inline'owanie małych funkcji. Zmniejsza to koszt wywołań i wielkość kodu.

3. Zmiana zmiennych na stałe, jeśli ich wartość jest znana w trakcie kompilacji.

Dzięki tej operacji kod wykonywalny zmniejszy się z 20 MB do 1.5-2.5MB[13].

8.2.2.3. Pełna optymalizacja

Kod, który jest tworzony przez Scala.js jest zgodny z restrykcjami narzuconymi przez kompilator Closure[14]. Jest to narzędzie stworzone przez firmę Google, które potrafi optymalizować kod Javascriptowy.[15] Celem tego kroku jest zmniejszenie rozmiaru pliku javascript i uczynienie konstrukcji w nim występujących bardziej wydajnymi. W wyniku tego procesu otrzymywany jest plik wykonywalny o rozmiarze między 150KB do kilkuset KB[13]. W celu użycia pełnej optymalizacji należy wywołać polecenie **FullOptJS** w sbt.

8.2.2.4. Pliki javascript

W wyniku działań optymalizatorów tworzone są poniższe pliki javascript. Należy mieć na uwadze fakt, iż pliki bibliotek muszą zostać dołączone do kodu strony przed plikiem z kodem wykonywalnym.

Dodać nagłówki do tabeli wyjasniające co zawiera dana kolumna

Typ kompilacji	Wytworzony plik	Zawartość pliku
FastOptJS	scala-js-[Nazwa-Projektu]-fastopt.js	Plik z kodem wykonywalnym
FastOptJS	scala-js-[Nazwa-Projektu]-fastopt.js.map	Plik mapujący kod Scali do kodu Javascript.
FastOptJS	scala-js-[Nazwa-Projektu]-jsdeps.js	Plik z kodem zewnętrznych bibliotek użytych procesie tworzenia aplikacji
FullOptJS	scala-js-[Nazwa-Projektu]-opt.js	Plik z kodem wykonywalnym
FullOptJS	scala-js-[Nazwa-Projektu]-opt.js.map	Plik mapujący kod Scali do kodu Javascript.
FullOptJS	scala-js-[Nazwa-Projektu]-jsdeps.min.js	Plik z kodem zewnętrznych bibliotek użytych procesie tworzenia aplikacji.

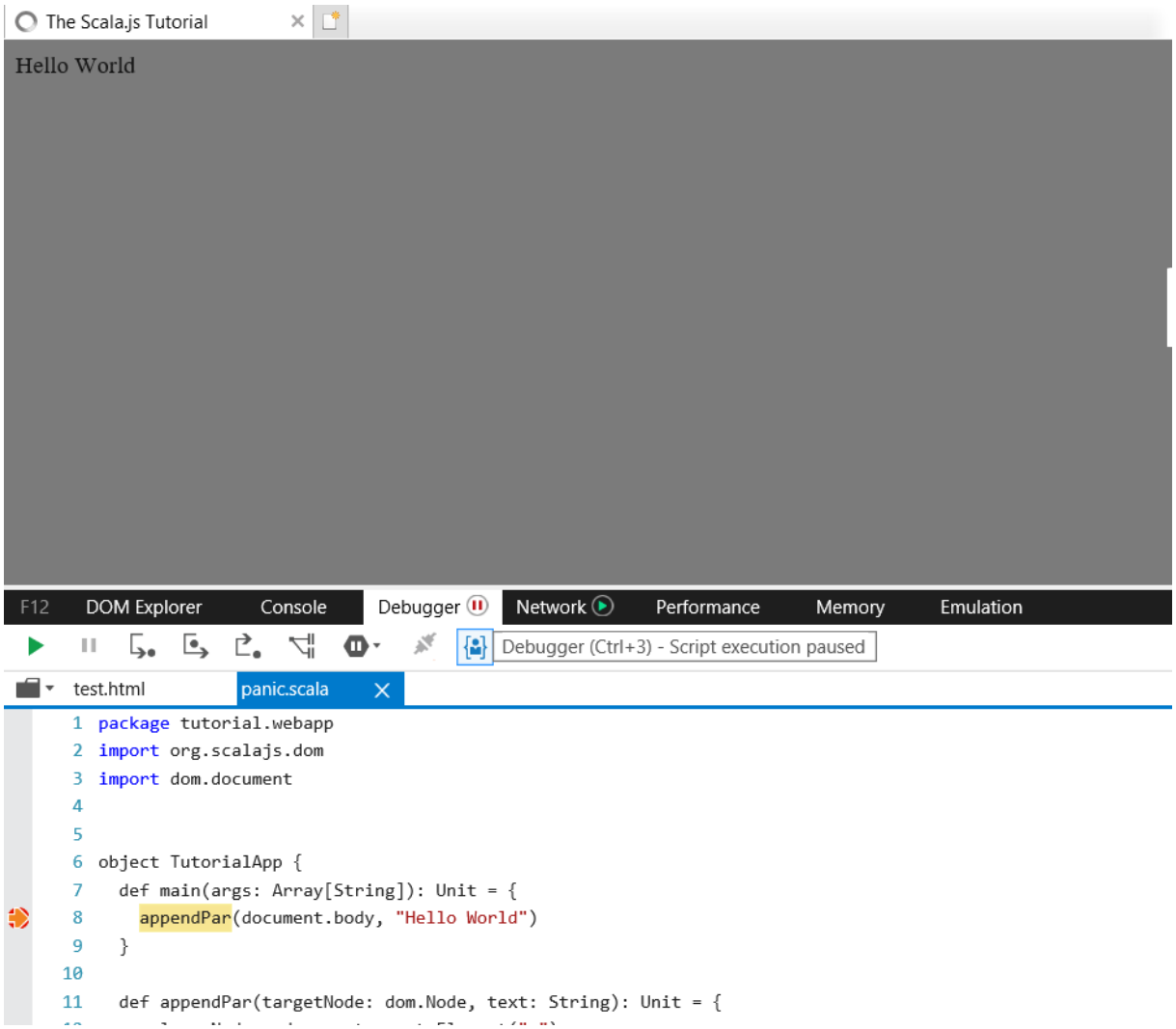
8.2.2.5. Debugowanie

Jednym z najważniejszych procesów, które na celu mają znalezienie błędów w powstałym kodzie jest debugowanie. Proces debugowania aplikacji powinien umożliwiać programiście przynajmniej możliwość zatrzymania kodu w dowolnym miejscu, podejrzenia kodu źródłowego jak i możliwość odczytu i modyfikacji zmiennych. Kod stworzony za pomocą kompilatora scala.js jest bardzo łatwy w debugowaniu. Przeglądarki posiadają możliwość dołączenia mapy kodów źródłowych, które do danych linijek kodu javascript przypisują ich odpowiedniki w pliku źródłowym. Umożliwia to prace z oryginalnym kodem źródłowym podczas używania przeglądarki internetowej. Pliki z mapami źródłowymi mają format **{SkompilowanyPlik}.map**, gdzie SkompilowanyPlik jest artefaktem naszego procesu kompilacji, np. **scala-js-[Nazwa-Projektu]-fastopt.js**.

8.2.3. Hello World

Warto jest robić tą sekcję gdzie opisze proces tworzenia hello worlda? Nie będe tego aktualnie pisał bo tylko będe tracił czas jeśli jest to zbedne. Ale z chęcią bym to opisał. Czyli jakie pliki źródłowe utworzyć i podać minimalny kod aby odpalić kod.

[3] Na obrazku na stronie widnieje już napis Hello World. Jest to artefakt, który został stworzony przy poprzednim uruchomieniu strony i jeszcze się nie odświeżył.



Rys. 8.2. Przykład procesu debugowania kodu. Nasz program zostaje zatrzymany na linijce przed dodaniem napisu Hello World. [3]

9. Opis aplikacji

9.0.1. Założenia aplikacji

Aplikacja ma za zadanie nauczyć użytkownika słówek z danego języka obcego, który jest wybrany przez użytkownika. Program od strony serwerowej został napisany w języku C# z użyciem frameworka ASP.NET MVC z wykorzystaniem bazy danych Microsoft SQL. Od strony klienta użyty został HTML wraz z arkuszami styli stworzonymi w oparciu o style SASS. Uruchamiany kod Javascript był skompilowanym kodem S

todo[inline] Zauważyłem, że używam tych citów do : -zostawienia odnośnika dla czytelnika, jeśli chciałby o danym temacie poczytać więcej. Na przykład odnośnik do kompilatora Closure, gdy wspominam o tym, że scala.js spełnia wszystkie warunki do użycia kompilatora closure.

10. Podsumowanie oraz wnioski

TODO

Bibliografia

[1] Dr. Bruce Abbott's. *Human Memory*. URL: *http://users.ipfw.edu/abbott/120/Ebbinghaus.html* (term. wiz. 2017-12-31).

[2] Dr. John Wittman. *The Forgetting Curve*. URL: *https://www.csustan.edu/sites/default/files/groups/Writing%20Program/forgetting_curve.pdf* (term. wiz. 2017-12-31).

[3] *Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10)*. URL: *https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application* (term. wiz. 2017-12-31).

[4] *Ninject*. URL: *https://github.com/ninject/Ninject* (term. wiz. 2017-12-31).

[5] *SQL Server 2016*. URL: *https://www.microsoft.com/pl-pl/sql-server/sql-server-2016* (term. wiz. 2017-01-03).

[6] *What is NuGet?* URL: *https://www.nuget.org/* (term. wiz. 2017-01-03).

[7] Microsoft. *Unit Testing: Testing the Inside*. URL: *https://msdn.microsoft.com/en-us/library/jj159340.aspx* (term. wiz. 2017-01-01).

[8] *Syntactically Awesome Style Sheets*. URL: *http://sass-lang.com/* (term. wiz. 2017-01-01).

[9] *Scala (programming language)*. URL: *https://en.wikipedia.org/wiki/Scala_(programming_language)* (term. wiz. 2017-01-02).

[10] *Tour of Scala*. URL: *http://docs.scala-lang.org/tour/tour-of-scala.html)* (term. wiz. 2017-01-02).

[11] *Scala.js*. URL: *http://www.scala-js.org/* (term. wiz. 2017-01-02).

[12] *Hands-on Scala.js : The Compilation Pipeline*. URL: *http://www.lihaoyi.com/hands-on-scala-js/#TheCompilationPipeline* (term. wiz. 2017-01-02).

[13] *Compilation and optimization pipeline*. URL: *https://www.scala-js.org/doc/internals/compile-opt-pipeline.html* (term. wiz. 2017-01-02).

[14] *Understanding the Restrictions Imposed by the Closure Compiler*. URL: *https://developers.google.com/closure/compiler/docs/limitations* (term. wiz. 2017-01-02).

[15] *Closure Compiler*. URL: *https://developers.google.com/closure/compiler/* (term. wiz. 2017-01-02).

[16] Nick Fitzgerald John Lenz. *Source Map Revision 3 Proposal*. URL: *https://sourcemaps.info/spec.html* (term. wiz. 2017-01-02).