Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Bazy Danych

Sławomir Zabkiewicz
Nr albumu 5081

Kontener Inversion Of Control dla obiektów Ruby realizujący wzorzec projektowy Dependency Injection

Praca magisterska
Napisana pod kierunkiem
dra inż. Mariusza Trzaski

Warszawa, wrzesień, 2009

Streszczenie

Paradygmat Inversion of Control, czyli odwrócenie sterowania w projektach programistycznych polega przede wszystkim na przeniesieniu odpowiedzialności za kontrolę wybranych czynności na zewnątrz obiektu, komponentu. Najpopularniejszą obecnie realizacją tego paradygmatu jest wstrzykiwanie zależności - Dependency Injection.

Zagadnienie to jest bardzo popularne w środowisku programistów języka Java, gdzie funkcjonuje aktualnie kilka bardzo rozbudowanych implementacji wzorców projektowych przenoszących odpowiedzialność za tworzenie obiektów i ich łączenie do kontenerów Inversion of Control.

Poniższa praca koncentruje się na jednym z tych wzorców, a głównym jej celem było stworzenie implementacji Dependency Injection w skryptowym języku Ruby. Dostępne w nim techniki metaprogramowania oraz dynamiczny charakter samego języka pozwalają w prosty i przyjemny sposób stworzyć minimalistyczną wersję kontenera obiektów, który bardzo łatwo można rozszerzać o dodatkowe funkcje.

Ruby jest stale rozwijanym stosunkowo nowym językiem programowania. Ze względu na prostotę i czytelność programów oraz jednocześnie wszechobecną obiektową składnię daje bardzo wiele możliwości i dzięki temu zyskuje wielu zwolenników.

Stworzony na potrzeby tej pracy kontener dla obiektów Ruby stanowi istotne rozszerzenie możliwości języka i jednocześnie pozwala znacznie zmniejszyć kod oprogramowania. Jest to szczególnie ważne przy tworzeniu dużych zespołowych projektów programistycznych. Ułatwia szybką zmianę i zapewnia łatwą konserwację takich projektów.

W celu zaprezentowania poprawności działania kontenera Inversion of Control została przygotowana, również w języku Ruby, aplikacja obsługująca protokół XML-RPC. Użycie w tej aplikacji wstrzykiwania zależności oraz kontroli nad obiektami przez kontener pozwala programiście skoncentrować się na tym, co ważne – na implementacji samego protokołu.

Podziękowania

Autor składa serdeczne podziękowania dla Pana dra inż. Mariusza Trzaski za opiekę nad projektem, konsultacje i cenne uwagi przekazywane w trakcie tworzenia niniejszej pracy.

Spis treści:

1. Wstęp

W dzisiejszych czasach duża część oprogramowania pisana jest w celu zautomatyzowania pewnych rzeczywistych procesów. Zasadniczym utrudnieniem w tej dziedzinie jest sposób automatycznego połączenia ze sobą, wydawałoby się niezależnych elementów języków programowania, takich jak obiekty. Problem ten związany jest z paradygmatem odwrócenia sterowania (ang. Inversion of Control, IoC), który polega na przeniesieniu odpowiedzialności za kontrolę wybranych czynności na zewnątrz obiektu.

W opisywanym przypadku jest to czynność tworzenia powiązań pomiędzy obiektami. Ten przypadek odwrócenia sterowania nazywany jest wstrzykiwaniem zależności (ang. Dependency Injection, DI) i polega na tworzeniu połączeń między komponentami w postaci architektury plug-in.

W ostatnich latach powstało kilka niezależnych implementacji wzorca Dependency Injection w różnych językach programowania. Dla coraz bardziej popularnego języka Ruby istnieją dwa powiązane ze sobą rozwiązania, Needle oraz Copland. Są one jednak dosyć oporne i przeładowane nawykami z języka Java, które w tak innowacyjnym i dynamicznym środowisku jak Ruby nie powinny się znaleźć. W efekcie sam autor tych dwóch rozwiązań zachęca, aby ich nie używać.

1.1. Kontekst pracy

W drugim rozdziale opisane są zagadnienia stojące u podstaw paradygmatu Inversion of Control. Zaprezentowane zostały różne podejścia do odwrócenia sterowania, przy czym największy nacisk położony został na wzorzec projektowy Dependency Injection oraz sposoby jego implementacji. Rozdział zakończony jest szczegółową analizą porównawczą dwóch podstawowych typów Dependency Injection – Constructor Dependency Injection oraz Setter Dependency Injection.

Rozdział “Istniejące implementacje wzorca Dependency Injection” dotyczy zagadnienia Dependency Injection oraz przedstawia istniejące implementacje tego wzorca w różnych językach programowania. Nacisk położony został na rozwiązaniach w języku Java, ponieważ to właśnie w związku z nim narodziło się pojęcie wstrzykiwania zależności i aktualnie Java jest wiodącym językiem programowania ze względu na ilość dostępnych rozwiązań. Ponieważ część z nich stanowi odpowiedź na potrzeby oraz problemy ściśle związane z tą jedną technologią, w bardzo dobry sposób ukazują efektywność oraz sposób użycia Dependency Injection. Rozdział ten stanowi przegląd bibliotek realizujących wstrzykiwanie zależności począwszy od najwcześniejszych (Apache Avalon) poprzez najpopularniejsze (Spring, PicoContainer), aż po te nowoczesne oraz najbardziej zaawansowane (Google Guice).

Rozdział czwarty to opis rozwiązań implementacyjnych. Przedstawione zostały podstawy języka Ruby oraz jego zaawansowane elementy, które w dużym stopniu związane są z implementacją wzorca Dependency Injection. Szczególna uwaga zwrócona została na typy danych występujące w języku oraz sposób ich deklaracji. Najwięcej miejsca poświęcone zostało pojęciom obiektu i klasy oraz ich modyfikowaniu (metaprogramowanie).

W rozdziale 5 znajduje się opis implementacji wzorca Dependency Injection w języku Ruby. Zaprezentowane zostały sposoby oraz przykłady użycia zarówno kontenera obiektów

jak i obsługi zależności. Przykładem wykorzystania biblioteki jest opisany szczegółowo serwer XML-RPC, który realizuje ideę zdalnych obiektów poprzez wykorzystanie odwrócenia sterowania w postaci dostawców zarządzanych przez kontener.

W ostatnim rozdziale znajduje się podsumowanie pracy. Opisane są trudności napotkane podczas wykonywania pracy oraz wyciągnięte wnioski. Przede wszystkim zaprezentowane zostały możliwości dalszej rozbudowy zaimplementowanej biblioteki realizującej wzorzec Dependency Injection umożliwiający tworzenie środowiska typu enterprise w języku Ruby.

Ponieważ praca jest ściśle związana z językiem programowania Ruby większość przykładów kodu prezentowana jest w języku Ruby. Jedynie kod mający na celu ukazanie pewnych charakterystycznych zagadnień dla innych języków programowania oraz przykłady wykorzystania istniejących implementacji Dependency Injection prezentowane są w tych językach.

1.2. Cel pracy

Celem poniższej pracy jest przegląd dostępnych rozwiązań w dziedzinie paradygmatu Inversion od Control oraz stworzenie implementacji wzorca Dependency Injection dla języka Ruby, która będzie tak dynamiczna i prosta w użyciu jak sam język. Pośrednio autor niniejszej pracy postawił sobie za cel omówienie oraz usystematyzowanie informacji na temat metaprogramowania oraz zarządzania typami w języku Ruby. Dodatkowo, zaprezentowany przykład użycia biblioteki, serwer XML-RPC, stanowi nawiązanie do możliwości wykorzystania samego języka jak i implementacji Dependency Injection do tworzenia systemów rozproszonych.

1.3. Rezultat pracy

Rezultatem pracy jest w pełni funkcjonalna biblioteka implementująca wzorzec Dependency Injection w języku Ruby, umożliwiająca programistom proste wstrzykiwanie zależności przy użycia dostarczonego kontenera obiektów. Serwer XML-RPC stanowiący przykład użycia biblioteki to minimalistyczny serwer udostępniający funkcjonalność zdalnych obiektów. Obie aplikacje wykorzystują w pełni możliwości języka Ruby, a w szczególności jego technik metaprogramowania.

2. Inversion Of Control czyli odwracanie sterowania.

W świecie tworzenia oprogramowania, duży nacisk kładzie się na jego utrzymywanie. Zdarza się, że więcej czasu jest poświęcone na zarządzanie kodem, niż na jego pisanie. Wymaga to zarówno prawidłowego projektowania oprogramowania jak i dobrze zdefiniowanego procesu walidacji oraz testowania.

Testowanie kodu jest niezbędnym etapem tworzenia oprogramowania. Proces testowania ma zapewnić, że stworzony kod działa zgodnie z założeniami. Jest to tylko jeden z elementów wchodzących w skład procesu utrzymywania oprogramowania. Kolejnym sposobem jest tworzenie mniejszej ilości kodu bez utraty funkcjonalności. Im mniej kodu potrzeba do osiągnięcia celu, tym mniej kodu do utrzymywania [1]. Duża część instrukcji jest zbędna, ponieważ pozwala jedynie na przejście z jednego miejsca programu do drugiego. To właśnie z tym zbędnym kodem programiści i projektanci starają się sobie poradzić, używając mechanizmów opisanych w tej pracy.

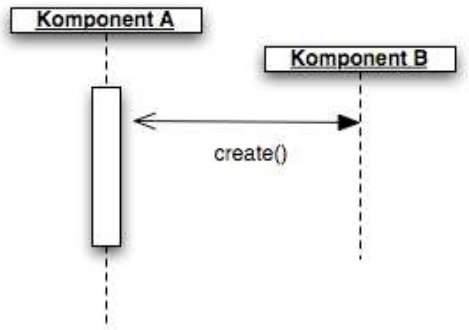
W rozdziale tym, zostaną dokładnie opisane zagadnienia paradygmatu Inversion of Control oraz wzorca projektowego Dependency Injection.

2.1. Inversion of Control

Przed przystąpieniem do opisu wzorca projektowego Dependency Injection, należy przedstawić paradygmat, który leży u podstaw powstania tego wzorca. Paradygmat ten nazywa się Inversion of Control (odwrócenie sterowania) i polega na przeniesieniu odpowiedzialności za kontrolę wybranych czynności na zewnątrz komponentu.

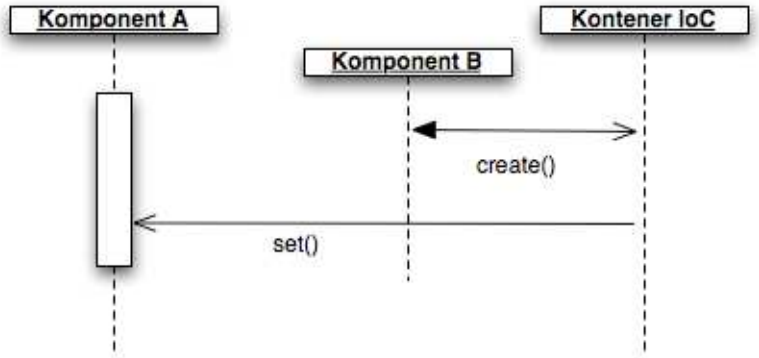
Te dwa pojęcia są bardzo często ze sobą mylone lub używane jako synonimy. Tymczasem Dependency Injection jest jedynie formą realizacji pewnej części odwrócenia sterowania. Inversion of Control jest pojęciem znacznie ogólniejszym, i jak zauważa Martin Fowler w [5] w przypadku implementacji kontenerów IoC, należy sobie zadać pytanie, jaki aspekt kontroli chcemy odwrócić. Najczęściej jest to tworzenie powiązań pomiędzy komponentami.

Tradycyjne podejście do programowania zakłada tworzenie zależności pomiędzy komponentami w nich samych jak pokazano na rysunku 1.



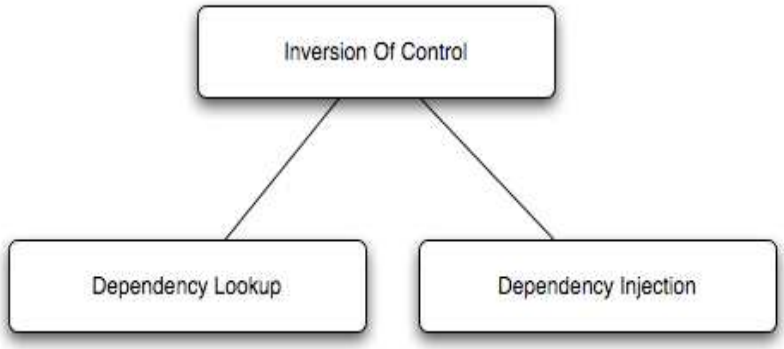
Rysunek 1. Tradycyjne podejście do komponentów programistycznych.
Źródło: Opracowanie własne na podstawie Fitech Laboratories Inc. Technical Whitepaper – xTier™ IoC-based System Configuration

Kontenery obiektów realizujące paradygmat Inversion of Control dostarczają komponentom mechanizmów dostępu do ich zależności (innych komponentów) w czasie ich cyklu życia, co ilustruje rysunek 2.



Rysunek 2. Podejście do komponentów zgodne z paradygmatem Inversion of Control.
Źródło: Opracowanie własne na podstawie Fitech Laboratories Inc Technical Whitepaper – xTier™ IoC-based System Configuration

Przez autorów *Pro Spring* [2] zaproponowany został podział usług realizowanych przez kontenery IoC przedstawiony na rysunku 3.



Rysunek 3. Podział usług Inversion of Control.
Źródło: Opracowanie własne

Dependency Lookup jest podejściem tradycyjnym i popularnym wśród programistów języka Java. Dependency Injection jest nowym wzorcem, przyjętym bardzo szybko i mimo, że wydaje się być mało intuicyjny na pierwszy rzut oka, niemniej jednak jest dużo bardziej elastyczny w użyciu niż Dependency Lookup. W przypadku Dependency Lookup, komponent musi pozyskać referencję do komponentu zależnego. W Dependency Injection zależności są wstrzykiwane do komponentów przez kontener Inversion Of Control.[2]

2.1.1 Dependency Lookup

Wzorzec Dependency Lookup może być zaimplementowany na dwa różne sposoby:

1. Dependency Pull
2. Contextualized Dependency Lookup (CDL)

Dependency Pull

Ten typ Dependency Lookup jest często używany przez programistów języka Java i polega na wyciąganiu zależności z rejestru, kiedy zachodzi potrzeba.

Przykład 1:

```
class Car

      def  initialize
            @registry = get_registry()
      end

      def run
            engine=@registry.get_object( engine )
            engine.start
      end
end
```

Źródło: Opracowanie własne

W przykładzie 1 w konstruktorze obiektu klasy **Car** pobierany jest pewien rejestr zależności przy pomocy metody **get_registry()**. Przy wywołaniu metody **run()** na obiekcie klasy Car pobierana jest z rejestru zależność – pewien komponent “engine”, który realizuje właściwą funkcjonalność dzięki swojej metodzie start().

Contextualized Dependency Lookup (CDL)

CDL jest podobne to Dependency Pull. Różni się jedynie tym, że wyszukiwanie zależności odbywa się w kontenerze, który zarządza danym komponentem, nie w centralnym rejestrze środowiskowym. Contextualized Dependency Lookup może być realizowane przez komponent implementujący interfejs.

Przykład 2:

```
module ManagedObject
      def lookup(registry,object_name)
            registry.get_object(object_name)
      end
end

class Message
      include ManagedObject

      def initialize(message)
            @message=message
      end
      def perform_lookup(registry)
            @writer=lookup(registry,'writer')
      end
      def run
            @writer.display(message)
      end
end
registry.add(new Message())
registry.managed_objects.each do |obj|
      obj.perform_lookup(registry)
end
message.run
```

Źródło: Opracowanie własne

Występują tu dwie klasy. Klasa **Message** implementuje moduł (interfejs) **ManagedObject**, dzięki któremu ma dostęp do metody **lookup** (container,object_name). Tworzony jest nowy obiekt klasy Message i dodawany do rejestru, w którym znajduje się już obiekt klasy Writer.

Następnie na wszystkich obiektach rejestru implementujących moduł ManagedObject wywoływana jest metoda perform_lookup, która w obiekcie klasy Message wywołuje metodę lookup w poszukiwaniu obiektu klasy Writer. Wywołanie metody run na obiekcie klasy Message de facto powoduje uruchomienie metody display na wcześniej pobranej zależności prowadzącej do obiektu klasy Writer.

2.1.2 Dependency Injection

Podobnie jak Dependency Lookup, Dependency Injection może być zaimplementowany na trzy różne sposoby:

1. Constructor Dependency Injection
2. Setter Dependency Injection
3. Interface Dependency Injection

Aby zrozumieć różnice pomiędzy Constructor Dependency Injection oraz Setter Dependency Injection należy zrozumieć różnice pomiędzy metodami a konstruktorami. Konstruktory posiadają zasadnicze ograniczenia narzucone im przez języki programowania (np.: konstruktor może być wywołany tylko raz). [3]

Constructor Dependency Injection

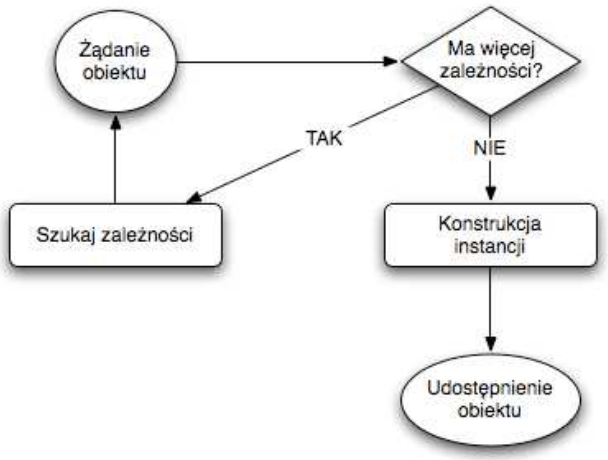
Zadaniem konstruktora jest zainicjowanie obiektu oraz jego początkowa konfiguracja przy użyciu dostarczonych do niego argumentów. Początkowa konfiguracja może polegać na wstrzykiwaniu zależności, co jest przedmiotem tej pracy lub na operacjach, które powinny zostać wykonane zanim obiekt zostanie użyty. Konstruktor posiada kilka ograniczeń, które odróżniają go od standardowych metod.

Konstruktor:

- nie zwraca żadnej wartości,
- może być wywołany tylko raz,
- jego nazwa jest wymuszona nazwą klasy.

Konstruktory dostępne są w większości obiektowych języków programowania jako sposób prawidłowego przygotowania obiektu do użycia.

Constructor Dependency Injection zakłada, że zależności, które mają zostać wstrzyknięte do komponentu dostarczone są w jego konstruktorze. Komponent posiada deklarację konstruktora lub listy konstruktorów przyjmujących argumenty będące jego zależnościami. Kontener Inversion of Control wstrzykuje te zależności do komponentu podczas jego inicjalizacji. Rysunek 4 przedstawia działania przeprowadzane kolejno przez kontener IoC w celu wykonania Constructor Dependency Injection. [3]



Rysunek 4. Schemat działania Constructor Dependency Injection.
Źródło: [3]

Przykład 3:

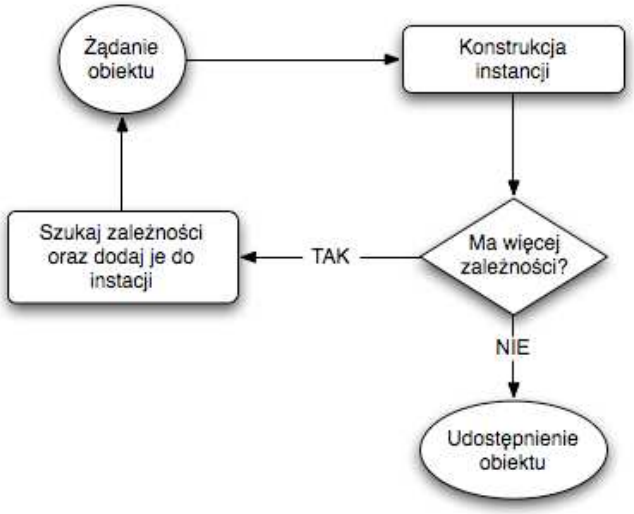
```
class Message
      attr_reader :writer
      def initialize(writer) @writer=writer end
end
```

Źródło: Opracowanie własne

W powyższym przykładzie, argumentem konstruktora klasy Message jest pewien obiekt writer. Podczas tworzenia obiektu klasy Message w jego konstruktorze następuje wstrzyknięcie zależności do atrybutu obiektu.

Setter Dependency Injection

W przypadku tego typu Dependency Injection, kontener Inversion of Control wstrzykuje zależności do komponentu używając do tego zadeklarowanych operacji dostępowych (tzw. settery). Settery komponentu określają zbiór zależności, którymi kontener Inversion of Control może zarządzać. Rysunek 5 przedstawia kroki, jakie muszą być wykonane, aby przygotować obiekt przy użyciu Setter Dependency Injection.



Rysunek 5. Schemat działania Setter Dependency Injection.
Źródło: [3]

Sekwencja wykonywanych operacji jest niemal identyczna jak w przypadku Constructor Dependency Injection. Jedynie operacja dostępowa (setter) jest użyta w miejscu konstruktora a wstrzykiwanie zależności odbywa się już po utworzeniu instancji komponentu.

Przykład 4:

```
class Message
      def set_writer(writer)
            @writer=writer
      end
end
```

Źródło: Opracowanie własne

Tym razem do wstrzyknięcia zależności – obiektu writer – użyta jest metoda dostępowa **set_writer**. Konieczność użycia powyższej metody przez kontener wymuszona jest przez przyjętą w danej implementacji Inversion Of Control konwencję nazw. W powyższym przykładzie jest to prefix set_ w nazwie metody dostępowej.

Interface Dependency Injection

Ten typ Dependency Injection jest bardzo podobny do Setter Dependency Injection. Zależności wstrzykiwane są przez metody dostępowe zadeklarowane w interfejsach. Każdy setter wydzielony jest do osobnego interfejsu.

Przykład 5:

```
module WriterInjectable
      def set_writer(writer)
            @writer=writer
      end
end

class Message
      include WriterInjectable
end
```

Źródło: Opracowanie własne

Klasa Message implementuje interfejs **WriterInjectable**, w którym zadeklarowana jest metoda **set_writer**. Klasa Message może implementować wiele interfejsów, z których każdy odpowiada za wstrzyknięcie jednej zależności:

Przykład 6:

```
class Message
      include WriterInjectable
      include LogInjectable
      include DBInjectable
end
```

Źródło: Opracowanie własne

2.2. Porównanie usług Inversion of Control

Dependency Injection jest zdecydowanie lepszym rozwiązaniem od Dependency Lookup. Jeżeli przyjrzymy się przykładom 3 oraz 4, wyraźnie widać, że użycie wstrzykiwania zależności nie ma żadnego wpływu na kod macierzystego komponentu.

Kod Dependency Pull (Przykład 1) jednocześnie aktywnie uczestniczy w wyszukiwaniu zależności w centralnym rejestrze. Z kolei CDL (Przykład 2) wymaga, aby klasy implementowały specyficzny interfejs a wyszukiwanie zależności musi być również zawarte w kodzie.

Przy użyciu Dependency Injection klasa komponentu powinna jedynie umożliwić wstrzykiwanie zależności przez kontener przy użyciu konstruktorów lub operacji dostępowych.

W tym przypadku istnieje możliwość użycia klas całkowicie niezależnie od kontenera Inversion of Control. Zależności mogą zostać dodane do komponentu ręcznie. W przypadku Dependency Lookup komponenty są bardzo mocno związane z kontenerem i trudno jest testować je w środowisku izolowanym. Testowanie w przypadku Dependency Injection jest proste, gdyż nic nie stoi na przeszkodzie, aby ręcznie dodać zależności do komponentu używając stosownego konstruktora lub operacji dostępowej.

Rozwiązania Dependency Lookup są, z konieczności, dużo bardziej złożone niż Dependency Injection. Wstrzykiwanie zależności upraszcza w dużym stopniu życie programisty. Pisany kod jest krótszy oraz prostszy. Jego tworzenie może być zautomatyzowane przez wiele dostępnych IDE.

Należy zwrócić uwagę, że przykłady 3 oraz 4 przedstawiają bierny kod, który nie próbuje aktywnie wykonać swojego zadania. To jest właśnie siła oraz prostota Dependency Injection, ponieważ bierny kod jest dużo prostszy do utrzymania niż ten aktywny.

Przykład 7 przedstawia metodę **perform_lookup**.

Przykład 7:

```
def perform_lookup(registry)
      @writer=lookup(registry,'writer')
end
```

Źródło: Opracowanie własne

W powyższym kodzie znajduje się wiele miejsc, w których może wystąpić błąd. Nazwa zależności **writer** może ulec zmianie. Instancja kontenera (**registry**) może być pusta (nil) lub otrzymana zależność może mieć niewłaściwy typ (w przypadku języków ze statyczną kontrolą typów). Użycie Dependency Lookup umożliwia podział aplikacji, ale wymaga dużej ilości dodatkowego kodu, aby połączyć jej komponenty na nowo, co znacznie utrudnia jej utrzymanie. [2]

2.3. Porównanie typów Dependency Injection

Najrzadziej używanym i najmniej wygodnym typem Dependency Injection jest Interface Injection. W przykładzie 5 i 6 widzieliśmy, ze jest dosyć rozwlekły i mało przyjemny w użyciu, ponieważ wymaga dużej ilości kodu.

Constructor Dependency Injection używany jest, kiedy konieczne jest, aby zależności zostały utworzone zanim komponent zostanie użyty. Większość kontenerów Inversion of Control dostarcza zbiór mechanizmów umożliwiających wstrzykiwanie zależności przy pomocy operacji dostępowych (settery), natomiast Constructor Injection jest metodą definiowania zależności do komponentu niezależnie od kontenera.

Jeżeli komponent udostępnia kontenerowi swoje zależności, ale może posiadać dla nich własne wartości domyślne, wtedy powinien zostać użyty Setter Dependency Injection.

Generalnie, różnice pomiędzy obydwoma typami Dependency Injection są analogiczne do różnic pomiędzy definiowaniem atrybutów komponentu poprzez konstruktor lub operację dostępową.

W przypadku, gdy chcemy, aby atrybut został zmodyfikowany tylko raz (przy tworzeniu komponentu) wtedy oznacza to, nie tylko, że komponent może polegać na swoich zależnościach finalnych w trakcie całego cyklu życia, ale również, że są one dostępne do użycia przy jego tworzeniu. Są to atrybuty stałe. Jest to forma tymczasowej hermetyzacji. W odniesieniu do wstrzykiwania zależności, podejście to może być przedstawione jako zamrażanie kompletnej i uformowanej już struktury obiektów. Można to bardzo łatwo przedstawić w języku Java na przykładzie deklarowania atrybutów finalnych. Przykład 8 przedstawia klasę **Atlas**.

Przykład 8:

```
public class Atlas {
      private final Earth earthOnBack;

       public Atlas(Earth earth) {
            this.earthOnBack = earth;
      } }
```

Źródło: [3]

Każda próba zmiany atrybutu earthOnBack wygeneruje wyjątek podczas fazy kompilowania. Jest to przydatna funkcjonalność gdyż zapewnia stały stan obiektu.

Stałość atrybutów nie jest możliwa przy użyciu Setter Dependency Injection. Operacje dostępowe nie różnią się niczym od zwykłych metod. Oznacza to, że w przypadku zachowania niezmienności zależności komponentu Constructor Dependency Injection jest lepszym wyborem.

Kolejnym problemem związanym z operacjami dostępowymi jest ich szybki przyrost. Jeżeli komponent posiada kilka zależności, tworzenie settera dla każdej z nich zaowocuje dużą ilością powtarzalnego kodu.

Z drugiej strony konstruktory przyjmujące wiele argumentów są bardzo nieczytelne. Istnienie wielu argumentów tego samego typu może być mylące, ponieważ jedyną rzeczą, która je odróżnia jest ich kolejność.

Przykład 9 prezentuje klasę Person w języku Java.

Przykład 9:

```
public class Person {
      private String firstName;
      private String lastName;
      private String nickName;
      public Person(String name1, String name2, String name3)
      { .. }
}
new Person(”John”,”Doe”,”johhny”);
```

Źródło: Opracowanie własne

Ponieważ wszystkie atrybuty klasy Person są typu String, najmniejszy błąd w nazewnictwie lub kolejności argumentów konstruktora zaowocuje błędnym ustawieniem zależności. W przypadku Setter Dependency Injection wstrzykiwanie zależności jest przejrzyste.

Przykład 10:

```
Person person = new Person();
person.setFirstName(”John”);
person.setLastName(”Doe”);
person.setNickName(”johnny”);
```

Źródło: Opracowanie własne

W przykładzie 10 wyraźnie widać, że Setter Dependency Injection jest dużo bardziej wygodne, gdy operuje się na wielu zależnościach tego samego typu. Ma to swoje zastosowanie szczególnie w testach jednostkowych, kiedy istnieje konieczność ustawienia zależności ręcznie.

Kolejnym problemem związanym z Constructor Dependency Injection jest duża ilość permutacji zestawu zależności w komponencie. Oznacza to, że zależności mogą być wstrzykiwane w różny sposób. Może to prowadzić do niekontrolowanego przyrostu liczby konstruktorów. Przykład 11 przedstawia klasę **Amphibian** w języku Ruby.

Przykład 11:

```
class Amphibian
      attr_reader :gills
      attr_reader :lungs
      attr_reader :heart
      def initialize(heart, gills)
            @heart = heart
            @gills = gills
      end
      def initialize(heart, lungs)
            @heart = heart
            @lungs = lungs
      end
end
```

Źródło: Opracowanie własne

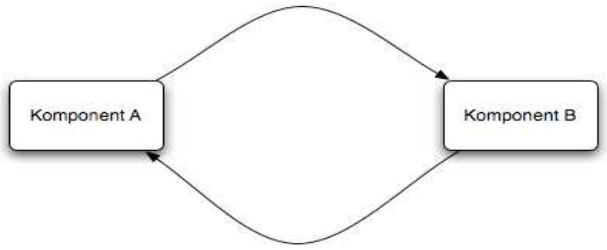
Kiedy potrzebujemy obiektu klasy **Amphibian** (płaz), który porusza się po wodzie, konstruujemy go przy użyciu skrzeli (**gills**). Na lądzie obiekt klasy Amphibian zostanie utworzony przy użyciu płuc (**lungs**). Serce (**heart**) jest wspólne dla obu przypadków.

W powyższym przykładzie potrzebujemy dwóch oddzielnych konstruktorów przystosowanych do każdego scenariusza. W przypadku większej ilości scenariuszy wzrośnie również liczba konstruktorów. Spowoduje to trudności w czytaniu kodu i rozróżnieniu poszczególnych konstruktorów. Zjawisko to nazywane jest “piramidą konstruktorów”.

Rozwiązaniem tego problemu jest Setter Dependency Injection, które pozwala zarządzać dowolną ilością permutacji zbioru zależności danego komponentu bez potrzeby pisania dodatkowego kodu.

Pętla referencyjna

Istnieje możliwość wystąpienia wzajemnej zależności pomiędzy komponentami. Rysunek 6 przedstawia taką relację.



Rysunek 6. Relacja kołowa pomiędzy komponentami w Constructor Dependency Injection.
Źródło: [3]

Taka symbiotyczna relacja pomiędzy dwoma komponentami często występuje w zależności rodzic-dziecko.

Przykład 12:

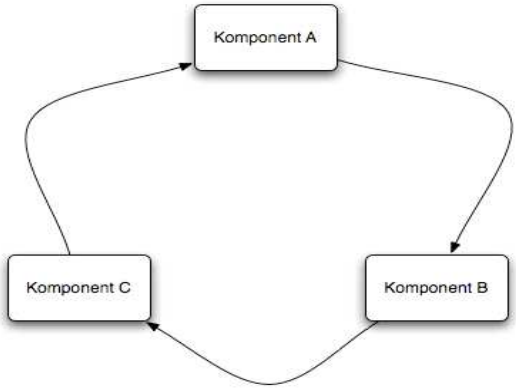
```
class Master
      attr_reader :slave

      def initialize(slave)
            @slave=slave
      end
end

class Slave
      attr_reader :master

      def initialize(master)
            @master=master
      end
end
```

Źródło: Opracowanie własne

W przykładzie 12 Master i Slave odwołują się do siebie nawzajem. Komponent **Master** posiada referencję do komponentu **Slave**, a komponent Slave z kolei posiada powrotną referencję do komponentu Master.

Jeżeli zdecydujemy się na utworzenie instancji klasy Master jako pierwszej, aby stan obiektu był poprawny, potrzebuje on instancji Slave jako zależności. W związku z tym jesteśmy zmuszeni utworzyć najpierw instancję klasy Slave. Niestety sytuacja się powtarza, ponieważ konstruktor klasy Slave wymusza istnienie instancji klasy Master. Jest to klasyczny problem jajko-kura. Zjawisko to nazywane jest “relacją kołową”.

Możemy utworzyć niebezpośrednią relację kołową, w której komponent A odwołuje się do komponentu B, komponent B odwołuje się do komponentu C, który z kolei wraca ponownie do komponentu A. Sytuację taką przedstawia rysunek 7.



Rysunek 7. Niebezpośrednia relacja kołowa pomiędzy komponentami.
Źródło: Opracowanie własne

Powyższy przypadek jest najczęściej spotykany. Oczywiście komponenty można dokładać bez ograniczeń dopóty dopóki ich końce się spotykają. Nie ma jasnej strategii mówiącej, która instancja powinna być utworzona jako pierwsza. Rozwiązaniem ponownie staje się Setter Dependency Injection.

Przykład 13 przedstawia przypadek, w którym dwa komponenty posiadają referencje do siebie nawzajem.

Przykład 13:

```
class Master
      attr_reader :slave

      def set_slave(slave)
            @slave=slave
      end
end

class Slave
      attr_reader :master

      def  set_master(master)
            @master=master
      end
end
```

Źródło: Opracowanie własne

W powyższym przykładzie klasy **Master** oraz **Slave** posiadają odpowiednie operacje dostępowe służące do utworzenia zależności pomiędzy nimi. Stosowna konfiguracja kontenera Inversion of Control pozwoli na zrealizowanie tych relacji. Obiekty obu klas zostaną utworzone przy użyciu ich domyślnych konstruktorów, które nie przyjmują żadnych argumentów. Następnie przy pomocy operacji dostępowych **set_slave** oraz **set_master** kontener utworzy wymagane zależności.

2.4. Podsumowanie

Wstrzykiwanie zależności poprzez metody dostępowe do komponentów jest zdecydowanie bardziej elastyczne od wstrzykiwania poprzez konstruktory, zwłaszcza, jeżeli nie ma konieczności tworzenia wszystkich relacji pomiędzy komponentami jednorazowo.

Niewątpliwą wadą Setter Dependency Injection, w przypadku języków ze statyczną kontrolą typów, jest brak możliwości utworzenia zależności stałych (finalnych).

Constructor Dependency Injection pozwala na tworzenie zależności finalnych, wymaga mniej kodu do napisania oraz zabezpiecza programistę przed przypadkowym nadpisaniem istniejących już zależności, do czego mogłoby dojść w przypadku użycia operacji dostępowych. Mimo tego, podejście takie stwarza jednak wiele niedogodności. W przypadku istnienia większej liczby zależności trudno jest zarządzać tak dużą ilością konstruktorów.

Dodatkowo w przypadku pętli referencyjnej, gdy dwa komponenty nawzajem od siebie zależą, niemożliwe jest wstrzyknięcie wzajemnych zależności przy pomocy Constructor Dependency Injection. Znakomicie w tej sytuacji sprawdza się natomiast wstrzykiwanie zależności przy pomocy metod dostępowych, ponieważ oba obiekty mogą zostać utworzone w postaci niekompletnej, a następnie połączone ze sobą.

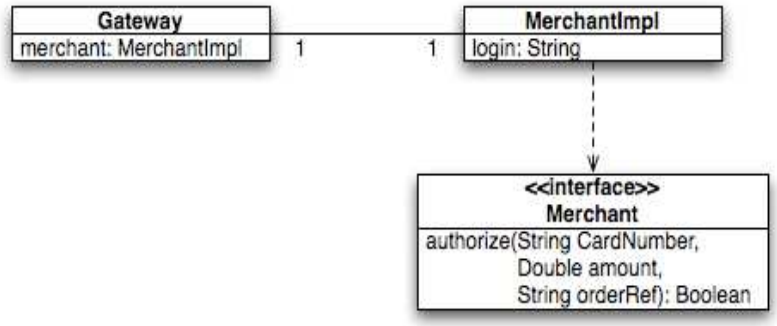
Wstrzykiwanie zależności opiera się na zarządzalnych oraz testowalnych komponentach zorganizowanych w wyraźnie wydzielone moduły, które mogą być składane, konstruowane oraz wdrażane w wielu różnych konfiguracjach. Umożliwia to programiście podział kodu na wydzielone oraz uporządkowane części. Taki podział ma ogromne znaczenie przy testowaniu oraz zarządzaniu kodem, zwłaszcza w przypadku dużych zespołowych projektów.

W proponowanej w tej pracy implementacji wzorca Dependency Injection wykorzystany został wariant **Setter Dependency Injection.** Wybór uzależniony był częściowo od technologii wykonania projektu. Język Ruby jest bardzo dynamicznym językiem i brak w nim mechanizmu stałych (finalnych) zależności. Dynamika ta powoduje, że wstrzykiwanie zależności poprzez metody dostępowe jest wygodnym sposobem łączenia komponentów i nie oznacza konieczności pisania większej ilości kodu.

3. Istniejące Implementacje Dependency Injection

Rozdział ten stanowi wprowadzenie do istniejących implementacji wzorca projektowego Dependency Injection. Skoncentrujemy się na rozwiązaniach w języku Java, ponieważ to właśnie w związku z nim narodziło się to pojęcie i aktualnie Java jest wiodącym językiem programowania ze względu na ilość dostępnych rozwiązań. Ponieważ część z nich stanowi odpowiedź na potrzeby oraz problemy ściśle związane z tą jedną technologią, w bardzo dobry sposób ukazują efektywność oraz sposób użycia Dependency Injection. Niniejszy rozdział stanowi przegląd bibliotek realizujących wstrzykiwanie zależności począwszy od najwcześniejszych (Apacze Avalon) poprzez najpopularniejsze (Spring, PicoContainer), aż po te nowoczesne oraz najbardziej zaawansowane (Google Guice).

W poniższym rozdziale przykłady użycia poszczególnych bibliotek implementujących wzorzec Dependency Injection oparte będą na bardzo prostym konkretnym problemie biznesowym. Rysunek 8 przedstawia diagram UML opisujący klasę Gateway, która reprezentuje bramkę płatności kartami kredytowymi w Internecie. Klasa MerchantImpl implementująca interfejs Merchant reprezentuje natomiast dostawcę usług autoryzujących karty kredytowe.



Rysunek 8. Diagram klas dla usługi płatności kartami kredytowymi.
Źródło: Opracowanie własne

Aby dokonać płatności należy wywołać metodę authorize na obiekcie klasy MerchantImpl. Atrybut login tej klasy oznacza ciąg znaków niezbędnych do uwierzytelniania w systemie dostawcy usługi płatności kartami kredytowymi. Przykład 14 prezentuje klasę **Gateway**.

Przykład 14:

```
public class Gateway {
	public MerchantImpl merchant;

	public Gateway(MerchantImpl merchant) {
		this.merchant=merchant;
	}
}
```

Źródło: Opracowanie własne

Bez użycia Dependency Injection zależność w postaci obiektu klasy **MerchantImpl** musi być ręcznie dodana przy tworzeniu obiektu klasy **Gateway** jako argument konstruktora.

Przykład 15 przedstawia interfejs **Merchant** oraz implementującą go klasę **MerchantImpl**.

Przykład 15:

```
public interface Merchant {
      Boolean authorize(Double amount, String orderRef);
}

public class MerchantImpl implements Merchant {
      String login;
      Double amount;
      String orderRef;

      public MerchantImpl(String login) {
            this.login=login;
            XmlRpc.setDriver("SAXParser");
      }

      public Boolean authorize(String cardNumber, Double, amount,
String orderRef){
            XmlRpcClient client = new XmlRpcClient(this.login);

            Vector params = new Vector( );
            params.addElement(cardNumber);
            params.addElement(amount);
            params.addElement(orderRef);

            Integer responseCode = (String)
            client.execute(„Payment.authorize”,params);

            if(responseCode==100)
                  return true;
            else
                  return false;
      }
}
```

Źródło: Opracowanie własne

Klasa **MerchantImpl** reprezentuje dostawcę usługi płatności kartami kredytowymi. Implementuje interfejs Merchant realizując metodę authorize jako wywołanie XmlRPC na zewnętrznym API dostępnym na serwerze dostawcy. Klasa MerchantImpl nawiązuje połączenie z usługodawcą używając atrybutu **login**, który definiowany jest w konstruktorze. Aby dokonać płatności kartą kredytową należy dostawcy wysłać jej numer (**cardNumber**), kwotę płatności (**amount**) oraz unikalny identyfikator zamówienia (**orderRef**). Metoda authorize zwraca *true* w przypadku, gdy płatność się powiedzie oraz *false* w przypadku wystąpienia błędu.

Motywacją do użycia Dependency Injection jest automatyzacja konfiguracji dostawcy usługi płatności kartami kredytowymi (Merchant) dla naszej bramki (Gateway). Nie będziemy zajmować się implementacją interfejsu Merchant (jest ona zazwyczaj dostarczana przez usługodawcę). Jedynym atrybutem, który musimy konfigurować jest **login**, który choć jest wartością charakterystyczną dla każdego usługobiorcy, nie zmienia się na tyle często, aby definiować go za każdym razem.

Chcemy zautomatyzować konfigurację naszej bramki do płatności kartami kredytowymi używając do tego dostępnych implementacji wzorca Dependency Injection.

3.1. Apache Avalon

Biblioteką realizującą wzorzec Dependency Injection, która powstała najwcześniej jest Apache Avalon. Pojawiła się ona w 1999 roku, a w 2002 wydzieliła się z projektu Apache Jakarta. Stanowiła kompletny kontener aplikacji jeszcze przed pojawieniem się Java Enterprise Edition. Najbardziej znanym systemem opartym na Apache Avalon jest Apache James Server, który jest serwerem pocztowym obsługującym protokoły SMTP, POP3 oraz IMAP.

Avalon-Framework (często zwany również AF, lub AF4 – od 4 wersji) definiuje kilka interfejsów reprezentujących cykl życia komponentu, które dostarczają wspólny schemat tworzenia, inicjalizowania oraz konfigurowania komponentów. Avalon dostarcza wstrzykiwanie zależności poprzez omówiony w poprzednim rozdziale Contextualized Dependency Lookup.

Komponenty kontenera Avalon implementują pewien interfejs **Serviceable**, który przedstawiony jest w przykładzie 16.

Przykład 16:

```
public interface Serviceable {
      public void service(ServiceManager sm) throws ServiceException;
}
```

Źródło: Opracowanie własne

Komponent implementujący interfejs Serviceable, będzie używał dostarczonego obiektu klasy ServiceManager, aby uzyskać dostęp do innych komponentów istniejących w środowisku. Interfejs ServiceManager przedstawiony jest w przykładzie 17.

Przykład 17:

```
public interface ServiceManager {
      boolean hasService(String key);
            Object lookup(String key) throws ServiceException;
      void release(Object object);
}
```

Źródło: [9]

Implementacja interfejsu **Serviceable** oznacza, że komponent może używać innych komponentów, ale nie precyzuje tego, co komponent robi. W przykładzie 18 przedstawione zostało użycie Apache Avalon dla klasy MerchantImpl w naszym przykładzie opisującym płatności kartami kredytowymi.

Przykład 18:

```
public class MerchantImpl implements Merchant, Serviceable {
      public MerchantImpl() {
            XmlRpc.setDriver("SAXParser");      }

      public void service(ServiceManager sm) throws ServiceException
      {
            this.login = sm.lookup(„login”);
      }

      public Boolean authorize(String cardNumber, Double amount,
      String orderRef) { ... }
}
```

Źródło: Opracowanie własne

Klasa MerchantImpl implementuje teraz dwa interfejsy: **Merchant** oraz **Serviceable**. Atrybut login nie jest już konfigurowany przy pomocy konstruktora. Zajmuje się tym metoda service, która wyszukuje w kontenerze ServiceManager żądany atrybut i konfiguruje go.

Przykład 19 prezentuje klasę Gateway, która również implementuje interfejs Serviceable.

Przykład 19:

```
public class Gateway implements Serviceable {
      public MerchantImpl merchant;

      public Gateway() {}

      public void service(ServiceManager sm) throws ServiceException
      {
            this.merchant = sm.lookup(„merchant”);
      }
}
```

Źródło: Opracowanie własne

Klasa **Gateway** implementuje interfejs **Serviceable**, dzięki któremu ma dostęp do obiektu klasy **ServiceManager** oraz jego metody lookup. Jest ona wywoływana w metodzie service i jako parametr przyjmuję nazwę klasy, której zależność powinna zostać wstrzyknięta (w tym przypadku jest to Merchant).

Obiekt klasy ServiceManager jest odpowiedzialny za zarządzanie, przechowywanie oraz wyszukiwanie zależności pomiędzy komponentami. Jego tworzenie polega na wypełnieniu jego rejestru instancjami klas znajdujących się w środowisku, tak jak pokazuje to przykład 20.

Przykład 20:

```
public void test() throws Exception
{
      DefaultServiceManager sm = new DefaultServiceManager();
      String login = ”gateway.realex.com”;
      MerchantImpl merchant = new MerchantImpl();
      Gateway gateway  = new Gateway();

      sm.put(”login”, login );
      sm.put(”merchant”, merchant );
      sm.put(”gateway”, gateway);
      sm.makeReadOnly();

      merchant.service(sm);
      gateway.service( sm );
}
```

Źródło: Opracowanie własne

Przy pomocy metody **put** pary {dowolna nazwa obiektu, obiekt} są umieszczane w rejestrze. Następnie na obiektach merchant oraz gateway wykonywana jest metoda **service,** która wyszukuje oraz konfiguruje ich zależności.

3.2. Spring Framework

Spring Framework jest aktualnie najpopularniejszą implementacją wzorca Dependency Injection. Jest odpowiedzialny za jego popularyzację w środowisku programistów.

Przez bardzo długi czas Spring był synonimem wstrzykiwania zależności. Został stworzony przez zespół, którego główną postacią był Rod Johnson. Początkowo biblioteka miała rozwiązywać specyficzne problemy twórców przy dużych projektach enterprise.

W roku 2003 Spring został opublikowany jako projekt open source. Od tego czasu nastąpił jego gwałtowny rozwój oraz popularność. Spring Framework udostępnia zestaw abstrakcji, modułów oraz punkty integracji dla aplikacji enterprise, open source oraz bibliotek komercyjnych. Posiada wsparcie dla wstrzykiwania zależności poprzez konstruktory oraz operacje dostępowe. Ponadto dostarcza mechanizmy do zarządzania obiektami utworzonymi przez kontener Inversion of Control. Spring jest bardzo dobrze udokumentowany. Istnieje bardzo dużo publikacji na jego temat, co sprawia, że jest coraz bardziej popularny w środowisku programistów Javy.

Spring dostarcza implementację Dependency Injection jako podstawowy sposób łączenia ze sobą zależnych od siebie obiektów. W aplikacjach opartych na Spring Framework preferowane jest użycie Dependency Injection zamiast wyszukiwanie zależności poprzez Dependency Lookup. Mimo to, w wielu środowiskach Spring nie ma możliwości automatycznego łączenia ze sobą wszystkich komponentów używając Dependency Injection. Należy wtedy użyć wyszukiwania zależności (Dependency Lookup) w celu uzyskania dostępu do początkowego zbioru komponentów.

Niewątpliwą zaletą kontenera Inversion of Control dostępnego w Spring Framework jest możliwość pełnienia przez niego funkcji łączącej go z zewnętrznymi kontenerami realizującymi Dependency Lookup.

Najważniejszą częścią implementacji Dependency Injection w Spring Framework jest BeanFactory. Klasa BeanFactory jest odpowiedzialna za zarządzanie komponentami oraz ich zależnościami. Termin Bean oznacza w Spring komponent zarządzany przez kontener Inversion Of Control. Aby aplikacja mogła komunikować się z kontenerem Inversion of Control musi stworzyć oraz skonfigurować instancję klasy implementującej interfejs BeanFactory.

Budowa tej instancji polega na utworzeniu obiektów klasy implementującej interfejs BeanDefinition. Konfiguracja ta pozwala na przechowywanie informacji na temat komponentu oraz jego zależności.

Opis zależności może być również przechowywany w zewnętrznym pliku konfiguracyjnym. Każda klasa BeanFactory, która dodatkowo implementuje interfejs BeanDefinitionRegistry pozwala na odczyt takiego pliku. Spring dostarcza w tym celu klasy PropertiesBeanDefinitionReader oraz XmlBeanDefinitionReader. Każdy Bean w Spring Framework posiada swoją nazwę oraz wiele aliasów. Pobieranie obiektów z BeanFactory oraz wstrzykiwanie zależności odbywa się tylko i wyłącznie przy użyciu ich nazw.

Przykład 21 przedstawia symulację użycia BeanFactory dla klasy Gateway.

Przykład 21:

```
public class Gateway {
      MerchantImpl merchant;
      public Gateway() {
            BeanFactory factory = getBeanFactory();
            merchant = (MerchantImpl) factory.getBean("merchant");
```

```
        }
        private static BeanFactory getBeanFactory() throws Exception {
              DefaultListableBeanFactory fac = new
              DefaultListableBeanFactory();
              PropertiesBeanDefinitionReader rdr = new
              PropertiesBeanDefinitionReader(fac);

              Properties props = new Properties();
              props.load(new FileInputStream(".beans.properties"));

              rdr.registerBeanDefinitions(props);
              return fac;
        }
  }
```

Źródło: [9]

W przykładzie 21 użyty został **DefaultListableBeanFactory** – jedna z dwóch domyślnych implementacji interfejsu BeanFactory dostępnych w Spring Framework.

Konfiguracja wczytywana jest z zewnętrznego pliku .**beans.properties** przy użyciu obiektu klasy **PropertiesBeanDefinitionReader.** Kiedy implementacja BeanFactory jest już stworzona oraz skonfigurowana pobierany jest obiekt klasy Merchant przy użyciu metody **getBean**. Metoda ta przyjmuje jako argument nazwę komponentu zdefiniowaną w pliku konfiguracyjnym.

Pliki properties są bardzo wygodne przy tworzeniu prostych aplikacji. W przypadku dużej ilości komponentów, zarządzanie ich konfiguracją może stać się bardzo uciążliwe. Z tego powodu Spring dostarcza klasę XmlBeanDefinitionReader, która pozwala na zarządzanie konfiguracją komponentów przy użyciu plików XML.

Przykład 22 przedstawia klasę MerchantImpl oraz użycie XmlBeanFactory, która wywodzi się z klasy DefaultListableBeanFactory i w prosty sposób rozszerza jej funkcjonalność o automatyczne wczytywanie konfiguracji komponentów przy użyciu XmlBeanDefinitionReader.

Przykład 22:

```
  public class MerchantImpl implements Merchant {
        String login;
        public MerchantImpl() {
              String file = ”beans.xml”;
              XmlBeanFactory fac = new XmlBeanFactory(new
              FileSystemResource(file));

              login = (String) fac.getBean(”login”);
        }
  }
```

Źródło: Opracowanie własne

XmlBeanFactory przyjmuje, jako parametr konstruktora, plik konfiguracyjny komponentów **beans.xml**. Kiedy implementacja BeanFactory jest już stworzona oraz skonfigurowana pobierany jest komponent **login** typu String przy użyciu jego nazwy.

Przykład 23 przedstawia podstawową konfigurację XML dla BeanFactory. Każdy komponent jest zdefiniowany przy użyciu znacznika <bean>.

Posiada on dwa wymagane atrybuty: **id** oraz **class**. Atrybut id używany jest do nadania komponentowi domyślnej nazwy. Atrybut lass jest definicją klasy komponentu.

Przykład 23:

```
<!DOCTYPE beans PUBLIC ”-//SPRING//DTD BEAN//EN”
”http://www.springframework.org/dtd/spring-beans.dtd”>
<beans>
      <bean id=”merchant” class=”pl.edu.pjwstk.MerchantImpl” />
      <bean id=”gateway” class=”pl.edu.pjwstk.Gateway” />
</beans>
```

Źródło: Opracowanie własne

Przykład 23 opisuje konfigurację dwóch komponentów **merchant** oraz **gateway**. Ich użycie przedstawia natomiast przykład 24.

Przykład 24:

```
public class Payment {
      public static void main(String[] args) throws Exception {
            BeanFactory factory = getBeanFactory();
            MerchantImpl merchant = (MerchantImpl)
            factory.getBean("merchant");

            Gateway gateway = (Gateway) factory.getBean("gateway");

            gateway.setMerchant(merchant);

            gateway.merchant.authorize(“4000000000000001”,99.95,”1”);
      }

      private static BeanFactory getBeanFactory() throws Exception {
            String file = “beans.xml”;

            XmlBeanFactory fac = new XmlBeanFactory(new
            FileSystemResource(file));

            return fac;
      }
}
```

Źródło: Opracowanie własne

Zauważmy, że oba komponenty – **merchant** oraz **gateway** są pobierane z kontenera Inversion Of Control przy pomocy metody **getBean()**. Dopiero później następuje ich połączenie przy pomocy metody dostępowej **setMerchant()** na obiekcie klasy **Gateway**. Jest to przykład wstrzykiwania zależności. Spring potrafi odczytywać konfigurację zależności z pliku konfiguracyjnego.

Przykład 25 przedstawia sposób konfiguracji zależności pomiędzy komponentami. Aby zdefiniować Setter Dependency Injection, należy do znacznika <bean> wstawić znacznik <property>.

Przykład 25:

```
<bean id=”gateway”  class=”pl.edu.pjwstk.Gateway” >
      <property name= gatewayMerchant >
            <ref local= ”merchant” />
      </property>
<bean>
```

Źródło: Opracowanie własne

Komponent **merchant** został przypisany do właściwości **gatewayMerchant.** Znacznik **<ref>** łączy zależny komponent z właściwością (property).

Mając tak skonfigurowane zależności pomiędzy komponentami przykładowa aplikacja wyglądać będzie tak jak w przykładzie 26.

Przykład 26:

```
public class Payment {

      public static void main(String[] args) throws Exception {
            BeanFactory factory = getBeanFactory();
            Gateway gateway = (Gateway) factory.getBean("gateway");
            gateway.merchant.authorize(“4000000000000001”,99.95,”1”);

      }

      private static BeanFactory getBeanFactory() throws Exception {
            String file = ”beans.xml”;
            XmlBeanFactory fac = new XmlBeanFactory(new
            FileSystemResource(file));
            return fac;
      }
}
```

Źródło: Opracowanie własne

W tym przypadku nie ma konieczności ręcznego zarządzania obiektem klasy MerchantImpl. Kontener Inversion of Control sam zarządza zależnościami pomiędzy komponentami czytając konfigurację BeanFactory z pliku konfiguracyjnego.

Spring Framework oprócz Setter Dependency Injection dostarcza również funkcjonalność do wstrzykiwania zależności przy użyciu konstruktorów. Konstruktor przyjmuje argumenty dla każdej zależności, która powinna zostać połączona z danym komponentem. Wadą tego rozwiązania (jak opisano w Rozdziale 2) jest brak możliwości zmiany tak wprowadzonej zależności w późniejszym czasie.

Przykład 27 przedstawia klasę Merchant, której konstruktor pozwala na definicję jej zależności.

Przykład 27:

```
public class MerchantImpl implements Merchant {
String login;

      public MerchantImpl(String login) {
            this.login=login;
            XmlRpc.setDriver("SAXParser");
      }

      public Boolean authorize(String cardNumber, Double amount,
      String orderRef) {
            ...
      }
}
```

Źródło: Opracowanie własne

Jak widać, nie da się utworzyć instancji klasy MerchantImpl bez podania wartości argumentu login. Spring dostarcza mechanizmów, które pozwalają wstrzykiwać zależności przy pomocy konstruktora na podstawie definicji komponentu w pliku konfiguracyjnym.

Definicja Constructor Dependency Injection dla klasy Merchant przedstawiona jest w przykładzie 28.

Przykład 28:

```
<bean id="provider" class="pl.edu.pjwstk.MerchantImpl">
      <constructor-arg>
            <value>gateway.realex.com</value>
      </constructor-arg>
</bean>
```

Źródło: Opracowanie własne

Przy konfiguracji Constructor Dependency Injection nie jest używany znacznik <property> (jak w przypadku operacji dostępowych). Aby zdefiniować wstrzykiwanie zależności przez konstruktor używany jest znacznik **<constructor-arg>.** Ponieważ argument konstruktora nie jest innym komponentem jego wartość zdefiniowana jest poprzez znacznik **<value>.** W przypadku większej ilości argumentów konstruktora, należy dla każdego znacznika <constructor-arg> podać atrybut XML „index", który oznacza pozycję argumentu na liście parametrów konstruktora. Sytuacja taka została zaprezentowana w przykładzie 29.

Przykład 29:

```
<bean id="provider" class="pl.edu.pjwstk.MerchantImpl">

      <constructor-arg index="0">
            <value>gateway.realex.com</value>
      </constructor-arg>

      <constructor-arg index="1">
            <value>8080</value>
      </constructor-arg>

      <constructor-arg index="2">
            <value>true</value>
      </constructor-arg>
</bean>
```

Źródło: Opracowanie własne

Tym razem konstruktor przyjmuje 3 argumenty i każdy z nich jest oznaczony odpowiednim atrybutem index:

- Login (adres bramy) – index 0
- Port (port, na którym nasłuchuje serwer HTTP) – index 1
- SSL (czy komunikacja jest szyfrowana) – index 2

W przypadku, gdy klasa posiada więcej konstruktorów o tej samej liczbie argumentów, może się okazać, że Spring nie będzie w stanie zdecydować, który konstruktor powinien zostać użyty.

Aby uniknąć takiej sytuacji należy podać w definicji komponentu, jakiego typu są poszczególne argumenty konstruktora. Przykład 30 pokazuje sposób użycia atrybutu "type" w definicji argumentu konstruktora.

Przykład 30:

```
<bean id="provider" class="pl.edu.pjwstk.Merchant">
      <constructor-arg index="0" type="string">
            <value>gateway.realex.com</value>
      </constructor-arg>
</bean>
```

Źródło: Opracowanie własne

Wzorzec Dependency Injection zapewnia, że programista nie musi tworzyć obiektów ręcznie w kodzie swojego programu. Zamiast tego wystarczy, że opisze jak obiekty powinny zostać utworzone oraz jakie zależności powinny zostać z nimi połączone.

Spring Framework znakomicie sobie radzi z tym zadaniem przy pomocy plików konfiguracyjnych a kontener Inversion of Control odpowiada za tworzenie oraz wstrzykiwanie zależności do opisanych przez programistę komponentów.

Spring Framework dobrze sprawdza się w codziennym użyciu. Pozwala na szybsze i prostsze składanie aplikacji w całość. Duża ilość dokumentacji oraz liczne środowisko związane z projektem zapewniają mu dużą popularność wśród programistów języka Java. Biblioteka jest używana z powodzeniem w wielu dużych międzynarodowych projektach enterprise.

3.3. PicoContainer

PicoContainer był pierwszą biblioteką wspierającą Constructor Dependency Injection. Został udostępniony jako lekki system pozwalający łączyć ze sobą komponenty programistyczne. Oprócz Constructor Dependency Injection PicoContainer posiada również wsparcie dla wstrzykiwania zależności poprzez metody dostępowe, jednak twórcy biblioteki wyraźnie odradzają jego użycie na rzecz wstrzykiwania poprzez konstruktory. PicoContainer jest uznawany za dosyć uciążliwy w konfiguracji oraz integracji. Dlatego też powstał siostrzany projekt Nano Container, który ma wypełnić lukę w warstwie konfiguracji pozostawiając wstrzykiwanie zależności na barkach PicoContainer.

W przykładzie 31 PicoContainer używa konstruktora, aby wstrzyknąć zależność – instancję klasy MerchantImpl do klasy Gateway. Klasa **Gateway** musi definiować konstruktora, którego argumentami są jej zależności.

Przykład 31:

```
public class Gateway {
      private MerchantImpl merchant;
      public Gateway(MerchantImpl merchant) {
            this.merchant = merchant;
      } }
```

Źródło: Opracowanie własne

Klasa Merchant również jest zarządzana przez Pico Container. Jej zależnością będzie login. Klasa Merchant widoczna jest w przykładzie 32.

Przykład 32:

```
public class MerchantImpl implements Merchant {
      String login;
      public MerchantImpl(String login) {
            this.login=login;
            XmlRpc.setDriver("SAXParser");
      }
      public Boolean authorize(String cardNumber, Double amount,
      String orderRef) {
            ...
      } }
```

Źródło: Opracowanie własne

Następnie Pico Container musi wiedzieć, które zależności ze sobą połączyć. Przykład 33 pokazuje sposób łączenia ze sobą komponentów. Zależności rejestrowane są w obrębie kontenera - instancji jednej z kilku dostarczanych w bibliotece klas. W tym przypadku jest to domyślna klasa **DefaultPicoContainer**.

Przykład 33:

```
MutablePicoContainer pico = new DefaultPicoContainer();

Parameter[] params =  {new ConstantParameter("gateway.realex.com")};

pico.registerComponentImplementation(„MerchantImpl”, params);

pico.registerComponentImplementation(„Gateway” , Gateway.class);
```

Źródło: Opracowanie własne

Komponenty rejestrowane są w kontenerze przy pomocy metody **registerComponentImplementation.** Argumenty będące typami prostymi grupowane są w tablice jako obiekty klasy **ConstantParameter**.

PicoContainer nie posiada funkcjonalności pozwalającej odczytywać konfigurację zależności z pliku zewnętrznego. Siostrzany projekt – NanoConainer dostarcza odpowiednie wrappery, które umożliwiają odczytanie konfiguracji z pliku XML. NanoContainer parsuje plik konfiguracyjny a następnie konfiguruje odpowiedni kontener PicoContainer.

3.4. Google Guice

Guice jest relatywnie nową biblioteką realizującą wzorzec Dependency Injection. Stworzona została przez zespół Bob’a Lee w firmie Google. Biblioteka wykorzystuje zalety piątej wersji języka Java mocno podkreślając bezpieczeństwo typów danych.

Jej twórcy preferują zwięzłość oraz rygorystyczne zasady konfiguracji. Guice jest powszechnie używane w firmie Google, szczególnie w bardzo obszernym projekcie AdWords. Wersja Google Guice znalazła się również w sercu frameworka stron www – Structs2. Biblioteka jest również dostępna w Google Web Toolkit (GWT) pod nazwą Gin.

Google Guice posiada wsparcie zarówno dla wstrzykiwania zależności poprzez konstruktory jak i operacje dostępowe. Ponadto posiada wiele interesujących alternatyw dla tych dwóch domyślnych typów Dependency Injection.

W przypadku Guice nie trzeba tworzyć specjalnych fabryk, aby łączyć komponenty ze sobą, wystarczy bardzo mała ilość konfiguracji, która widoczna będzie w obrębie całej aplikacji. Przykład 34 przedstawia najprostszą konfigurację Dependency Injection wprowadzoną bezpośrednio w kodzie.

Przykład 34:

```
public class Gateway {
      private final MerchantImpl merchant;
      @Inject
      public Gateway (MerchantImpl merchant) {
            this.merchant = merchant;
      }
}
```

Źródło: Opracowanie własne

Adnotacja **@Inject** pokazuje kontenerowi miejsce, w którym należy wstrzyknąć zależności. W tym przypadku będzie to konstruktor klasy Gateway przyjmujący jako argument obiekt klasy Merchant. @Inject może również zostać użyta przy normalnych metodach oraz atrybutach. Przykład 35 przedstawia sposób użycia adnotacji **@Inject** dla zwykłego atrybutu login klasy **MerchantImpl**.

Przykład 35:

```
public class MerchantImpl  implements Merchant{
      @Inject @Named(“login”)
      String login;
      public MerchantImpl () {
            XmlRpc.setDriver("SAXParser");
      {
      public Boolean authorize(String cardNumber, Double amount,
      String orderRef)
      { ... }
}
```

Źródło: Opracowanie własne

Obok adnotacji @Inject pojawiła się adnotacja **@Named**, która może być użyta wielokrotnie. Adnotacja @Named używa identyfikatora znakowego do odróżnienia wielu punktów wstrzykiwania zależności.

Tabela 1 przedstawia typy Dependency Injection dostępne w Google Guice oraz kolejność ich wykonywania przez kontener Inversion of Control.

Typ Dependency Injection	**Kolejność**	**Opis**
Konstruktor	Pierwszy	Tylko jeden możliwy
Atrybut	Drugi	Losowy wybór kolejności
Operacja dostępowa	Trzeci	Losowy wybór kolejności

Tabela 1. Typy Dependency Injection
Źródło: [1]

Tylko jeden konstruktor może być opatrzony adnotacją @Inject. Losowy wybór kolejności przy wstrzykiwaniu zależności w atrybutach i operacjach dostępowych oznacza, że programista nie może polegać na kolejności wstrzykiwania zależności przez kontener.

Wstrzykiwanie zależności z wykorzystaniem operacji dostępowych różni się od konwencji zastosowanej w Spring Framework. Tam, Setter Dependency Injection to operacje dostępowe o nazwach budowanych z prefixem „setXXX” (np. setMerchant), gdzie „XXX” to nazwa pojedynczej zależności, która ma zostać wstrzyknięta.

Guice odchodzi od tej konwencji nazw, dzięki czemu może operować na większej ilości argumentów w pojedynczej metodzie dostępowej. Powodem tego jest potrzeba uruchamiania przez kontener metod, które nie wstrzykują żadnej konkretnej zależności do obiektu, ale wykonują pewną początkową pracę (konfigurację) na rzecz tego komponentu.

Niezależnie od wybranego typu Dependency Injection, zakres widoczności nie ma znaczenia. Guice wykona swoją pracę wszędzie tam, gdzie napotka na adnotację @Inject i nie będzie istotne czy będzie to zakres private, protected czy public.

Konfiguracja kontenera odbywa się przy użyciu specjalnych klas implementujących interfejs **Module**. Przykład 36 przedstawia moduł konfigurujący zależności dla klasy Gateway.

Przykład 36:

```
public class GatewayModule implements Module {
      public void configure(Binder binder) {
            binder.bind(Merchant.class).to(MerchantImpl.class);
      } }
```

Źródło: Opracowanie własne

Klasę GatewayModule można również dziedziczyć z klasy AbstractModule zamiast implementować interfejs Module. Klasa AbstractModule implementuje ten interfejs i udostępnia metodę configure, która w tym przypadku nie przyjmuje żadnych argumentów wywołania. Przykład 37 przedstawia sposób użycia klasy AbstractModule.

Przykład 37:

```
public class GatewayModule extends AbstractModule {
      public void configure() {
            bind(Merchant.class).to(MerchantImpl.class);
      } }
```

Źródło: Opracowanie własne

Również klasa MerchantImpl posiada zależności w postaci atrybutu login. W przykładzie 38 przedstawiony jest sposób konfiguracji zależności tego typu.

Przykład 38:

```
public class MerchantModule extends AbstractModule {
      public void configure() {
            bindConstant()
            .annotatedWith(Names.named("login"))
            .to("gateway.realex.com");
      } }
```

Źródło: Opracowanie własne

Tym razem na binderze została użyta metoda **annotatedWith,** która konfiguruje zależności dla atrybutów klas oznaczonych adnotacjami @Inject oraz @Named.

Aby użyć tak skonfigurowanych zależności należy stworzyć instancję klasy Injector. Guice pobiera implementacje interfejsu Module i wstrzykuje zdefiniowane w nich zależności do komponentów. Aby stworzyć instancję klasy Injector można posłużyć się dostarczanymi przez Guice fabrykami, które są prostymi statycznymi klasami udostępniającymi funkcjonalność do tworzenia kontenerów. Przykład 39 przedstawia sposób tworzenia instancji klasy Injector oraz użycia jej do wstrzyknięcia zależności do zdefiniowanej wcześniej klasy Gateway.

Przykład 39:

```
public class Payment {
      public static void main(String[] args) {
            Injector injector= Guice.createInjector(new
            GatewayModule());
            Gateway gateway = i.getInstance(Gateway.class);
            Gateway.merchant.authorize(”4000000000000010”,99.95,”1”);
      } }
```

Źródło: Opracowanie własne

Metoda createInjector() przyjmuje zmienną liczbę argumentów, co pozwala na definiowanie zero lub więcej modułów opisujących konfigurację zależności. Nie ma jednak potrzeby podawania modułów opisujących konfigurację zależności dla komponentów, które same są zależnościami. W tym przypadku metoda **createInjector** nie musi być wywołana z obiektem klasy MerchantModule. Guice wstrzyknie te zależności automatycznie. Takie podejście pozwala na użycie obiektu klasy Injector na samej górze stosu aplikacji. Kontener utworzy graf obiektów w dół stosu.

3.5. Podsumowanie

Oprócz opisanych w powyższym rozdziale implementacji wzorca Dependency Injection, istnieje wiele innych zarówno w Javie jak i innych językach programowania. Niektóre z nich dostarczają dodatkowe funkcjonalności, ortogonalne do Dependency Injection lub koncentrują się na innym problemie związanym z odwróceniem sterowania.

Językowo C# oraz platforma .NET są podobne do Javy. Oba języki posiadają silną kontrolę typów, są obiektowe i muszą być kompilowane. Z tego względu problemy programistów C# są podobne do problemów pojawiających się podczas programowania w Javie. C# posiada porty bibliotek Spring oraz PicoContainer. Mimo to istnieją innowacyjne rozwiązania dla platformy .NET, takie jak Castle MicroKernel, które bardzo dobrze radzą sobie z problemami Dependency Injection. Kolejną biblioteką dla C# jest StructureMap, która prezentuje tradycyjne, podobne do Javy, podejście do wstrzykiwania zależności.

Dependency Injection jest trudne do implementacji w językach programowania takich jak C++. Głównym powodem jest brak refleksji. Mimo to możliwe jest wstrzykiwanie zależności poprzez użycie niestandardowych metod oraz narzędzi. Dla języka C++ biblioteki używają prekompilacji oraz automatycznego generowania kodu, aby umożliwić odwrócenie sterowania.

W realizacji wzorca Dependency Injection zaproponowanej w tej pracy wykorzystane zostały doświadczenia autora pracy z wymienionymi w tym rozdziale implementacjami.

Najbardziej wygodne oraz nowoczesne podejście zaprezentowane zostało w Google Guice. Mechanizm adnotacji jest bardzo prosty, ale równocześnie dostarcza ogromne możliwości.

Ze względu na technologię wykonania implementacji Dependency Injection będącej przedmiotem tej pracy nie ma możliwości skorzystania z mechanizmu adnotacji. Mimo to język Ruby jest na tyle dynamicznym językiem programowania, że posiada mechanizmy pozwalające na definiowanie zależności w komponentach na równie wysokim poziomie jak Google Guice. W tym przypadku do osiągnięcia wyznaczonego celu zostały wykorzystane opisane w następnym rozdziale techniki metaprogramowania.

4. Technologie wykorzystane przy tworzeniu pracy

4.1. Ruby – język programowania

Ruby jest skryptowym językiem programowania stworzonym w 1995 roku przez Yukihiro Matsumoto. Łączy w sobie to, co najlepsze w językach LISP, SmallTalk, Perl oraz Python. Ruby jest w pełni obiektowy, co oznacza, że wszystkie byty programistyczne są obiektami oraz rezultaty operacji na tych bytach również są obiektami. W ostatnich latach Ruby przeżywa gwałtowny wzrost popularności ze względu na powstanie Ruby on Rails - frameworku MVC służącego do tworzenia stron internetowych.

W poniższym rozdziale przedstawione zostaną podstawy języka, typy danych (opis przygotowany w oparciu o [6]) oraz jego zaawansowane elementy, które w dużym stopniu związane są z implementacją wzorca Dependency Injection w Ruby. Skoncentrujemy się przede wszystkim na pojęciu obiektu i klasy oraz na ich modyfikowaniu (metaprogramowanie).

4.1.1 Podstawy języka

Programy w języku Ruby mogą być pisane w 7-bitowym kodowaniu ASCII, kodowaniu Kanji lub w uniwersalnym UTF-8. Ruby jest językiem zorientowanym liniowo, co oznacza, że jego wyrażenia są parsowane do końca linii. Znak średnika (;) może zostać użyty do oddzielania wielu wyrażeń języka w jednej linii. Aby kontynuować wyrażenie w kolejnej linii należy wprowadzić znak backslash (\). Komentarze rozpoczynają się od znaku hash (#) i ciągną się do końca danej linii. [4]. Przykład 40 prezentuje proste wyrażenia w języku Ruby.

Przykład 40:

```
a = 19
b = 4; c=9
e = 8 + 18 \
       + 13
```

Źródło: Opracowanie własne

Istnieje możliwość wykomentowania bloków kodu przy użyciu =begin oraz =end. Odpowiada to użyciu kombinacji /* */ w języku Java. Przykład 41 prezentuje przykład wykomentowania bloku kodu.

Przykład 41:

```
=begin
      b = 6; c=9
      d = 5 + 8 + # nie potrzeba znaku '\'
      2
=end
```

Źródło: Opracowanie własne

Jeżeli parser języka napotka w kodzie programu na „__END__”, traktuje taką linię jako koniec programu. Każda następna linia nie będzie traktowana jako kod.

Typy danych

W języku Ruby wyróżnić można typy podstawowe, wbudowane oraz inne.

Liczby (Fixnum, Bignum, Float)

Liczby całkowite reprezentowane są w Ruby przez klasy Fixnum oraz Bignum. Liczby zmiennoprzecinkowe to klasy Float oraz BigDecimal. Odpowiednikiem klasy Float w Javie jest typ Double. W liczbach zmiennoprzecinkowych część ułamkowa oddzielana jestą od całkowitej kropką. Aby utworzyć wartość zmiennoprzecinkową wystarczy podać wartość dziesiętną liczby.

Łańcuchy znaków (String)

Ruby dostarcza kilka mechanizmów do tworzenia łańcuchów znaków. Każdy z nich tworzy obiekt klasy String. Przykład 42 przedstawia sposób tworzenia obiektów klasy String.

Przykład 42:

```
"Ruby is \"nice"\"    # => Ruby is "nice"
%Q!"Ruby is so nice"  # => "Ruby is so nice"
"It is #{Time.now}"   # => It is Sun Aug 16 20:19:03 +0200 2009
```

Źródło: Opracowanie własne

Zakresy (Ranges)

Zakresy tworzone są przy pomocy instrukcji „expr..expr” oraz „expr...expr”. Wersja z dwoma kropkami zakłada przedział zamknięty. Wersja z trzema kropkami zakłada przedział otwarty. Przykład 43 prezentuje sposób użycia zakresów:

Przykład 43:

```
(13..22)     # => 13..22
(12...22)    # => 12...21
```

Źródło: Opracowanie własne

Symbole (Symbol)

W języku Ruby, symbol jest identyfikatorem reprezentującym łańcuch znaków (nazwę). Symbole tworzone są poprzez wstawienie znaku „**:**” przed łańcuchem znaków.

Wyrażenia regularne (Regexp)

Wyrażenia regularne w Ruby są obiektami typu Regexp. Są one tworzone poprzez wywołanie konstruktora Regexp.new bądź, w krótkiej formie. Przykład 44 prezentuje wszystkie sposoby tworzenia wyrażeń regularnych.

Przykład 44:

```
/[0-9]+/
%r{[0-9]+}
Regexp.new('[0-9]+')
```

Źródło: Opracowanie własne

Tablice zwykłe (Array)

Obiekty klasy Array są tworzone przy pomocy wartości oddzieloych przecinkami i umieszczonych pomiędzy znakami [oraz].

Tablice asocjacyjne (Hash)

Obiekty klasy Hash tworzone są poprzez podanie par klucz => wartość pomiędzy znakami **{** oraz **}**.

Przykład 45 prezentuje prostą tablicę **atab** oraz tablicę asocjacyjną **harr**.

Przykład 45:

```
atab =['a','b','b']
atab[0]        # => 'b'
atab[0..1]     # => 'a','b'
atab[-1]       # => 'c'

harr = {1=>'a',2=>'b',5=>'c'}
harr[0]        # => nil
harr[1]        # => 'a'
harr[5]        # => 'c'
```

Źródło: Opracowanie własne

Nazwy

Nazwy w języku Ruby odnoszą się do stałych, zmiennych, metod, klas oraz modułów. Rodzaj obiektu określa pierwszy znak nazwy, który pomaga interpreterowi języka rozpoznać sposób użycia. Istnieje pewna liczba nazw zarezerwowanych dla języka Ruby i byty tworzone przez użytkownika nie mogą ich nadpisywać. Nazwy zmiennych tworzone są przez podanie ciągu znaków rozpoczynającego się od małej litery. Konwencja języka zaleca używanie podkreśleń '_' w łączeniu członów nazwy. Nazwy zmiennych instancji obiektów tworzone są poprzez dodanie prefixu '@' (np. @obj). Zaleca się tworzenie nazw zmiennych instancji wyłącznie z małych liter. Nazwy zmiennych klas tworzone są poprzez dodanie podwójnego prefixu '@@' (np. @@obj).

4.1.2 Duck Typing

Ruby jest dynamicznie typowanym (dynamically typed) językiem programowania. Oznacza to, że typy danych są określane w czasie wykonania programu. Mimo to, Ruby jest również silnie typowanym (strongly-typed) oraz bezpiecznym (type-safe) językiem, ponieważ nie pozwala na wykonanie operacji na argumentach, które mają nieprawidłowe typy oraz nie pozwala na operacje bądź konwersje, które prowadzą do błędnych rezultatów. Języki ze statyczną kontrolą typów, takie jak C# lub Java, potrafią wykryć błędy w fazie kompilacji. Ruby jest językiem skryptowym i jego charakter nie pozwala na statyczne wykrywanie błędów w kodzie. [4]

W statycznie typowanych językach programowania przyjęto założenie, że typem obiektu jest jego klasa. Oznacza to, że dla wszystkich instancji pewnej klasy ich typem jest właśnie ta klasa. Klasa definiuje metody, które obiekt posiada oraz jego stan (atrybuty), na których te metody operują. Przykład 46 prezentuje kod Java tworzący obiekt pewnej klasy CreditCard.

Przykład 46:

```
CreditCard card;
Card = holder.getCard("VISA");
```

Źródło: Opracowanie własne

Przykład ten deklaruje zmienną **card** typu **CreditCard** oraz ustawia jej wartość na obiekt karty kredytowej dla pewnego (wcześniej utworzonego) obiektu **holder** (właściciel karty).

Java wspiera koncepcję interfejsów, które są pewnego rodzaju zubożonymi abstrakcyjnymi klasami bazowymi. Klasa w języku Java może być zadeklarowana jako implementacja wielu takich interfejsów. Przykład 47 prezentuje użycie interfejsów.

Przykład 47:

```
public interface Card {
      String getNumber();
      String getExpiryDate();
}
public class CreditCard implements Card{
      public String getNumber() { ... }
      public String getExpiryDate() { ... }   }
```

Źródło: Opracowanie własne

Jak widać, nawet w Javie klasa nie zawsze jest typem. Zdarza się, że typ jest zbiorem klas a obiekty implementują wiele typów.

W języku Ruby, klasa nigdy nie jest typem obiektu. Typ obiektu jest definiowany poprzez jego zachowanie. Podejście to nazywane jest „duck typing". Nazwa wywodzi się ze słynnego zdania Dave'a Thomasa na temat typów w Ruby. Powiedział on, że: *„jeżeli obiekt chodzi jak kaczka, mówi jak kaczka oraz zachowuje się jak kaczka, wtedy interpreter potraktuje go jako kaczkę*".

Przykład 48 prezentuje klasę CreditCard w języku Ruby.

Przykład 48:

```
class CreditCard

      def initialize(holder,number)
            @holder = holder
            @number = number
      end

      def  append_card_to_file(file)
            file << @holder << ' ' << @number
      end
end
```

Źródło: Opracowanie własne

Aby zaprezentować „duck typing" przykład 49 prezentuje test dla klasy CreditCard w Javie.

Przykład 49:

```
require 'test/unit'
require 'card'

class TestAddCreditCard < Test::Unit::TestCase
      def test_add
            card = CreditCard.new(
            'Slawomir Zabkiewicz','4000000000000010')

            file = File.open('tmp','w')  do |file|
                  card.append_card_to_file(file)
            end
```

```
                file = File.open('tmp') do |file|
                      assert_equal(
                      'Slawomir Zabkiewicz','4000000000000010', f.gets)
                end
          ensure
                File.delete('tmp') if File.exists?('tmp')
          end
    end
```

Źródło: Opracowanie własne

Przykład ten pokazuje jak dużo pracy należy włożyć, żeby wykonać te proste operacje. Należy utworzyć plik, zapisać do niego dane, następnie otworzyć go ponownie i przeczytać jego zawartość, aby sprawdzić poprawność. Na koniec należy jeszcze usunąć plik, jeżeli istnieje. Jest to dosyć żmudne i czasochłonne.

Zamiast tego można wykorzystać „duck typing”. Wszystko, czego potrzebujemy to byt, który zachowuje się jak plik i do którego możemy zapisać pewną wartość. W tym przypadku wystarczy byt odpowiadający na metodę ‘<<’, która dopisuje zawartość do już istniejącej. Przykład 50 prezentuje ten sam test z użyciem obiektu klasy String.

Przykład 50:

```
    class TestAddCreditCard < Test::Unit::TestCase
          def test_add
                card = CreditCard.new(
                'Slawomir Zabkiewicz','4000000000000010')
                f = ''
                card.append_card_to_file(f)
                assert_equal(
                'Slawomir Zabkiewicz 4000000000000010',f)
          end
    end
```

Źródło: Opracowanie własne

Metoda klasy CreditCard uważa, że zapisuje dane do pliku. Zamiast tego dopisuje je do zwykłego łańcucha znaków. Mimo to test zakończy się sukcesem.

Nie musimy ograniczać się jedynie do łańcuchów znaków. Przykład 51 pokazuje, że dopisywanie do tablicy również spowoduje, że test zakończy się sukcesem.

Przykład 51:

```
    class TestAddCreditCard < Test::Unit::TestCase
          def test_add
                card = CreditCard.new(
                'Slawomir Zabkiewicz','4000000000000010')
                f =  []
                card.append_card_to_file(f)
                assert_equal(
                ['Slawomir Zabkiewicz',' ','4000000000000010'],f)
          end
    end
```

Źródło: Opracowanie własne

Jak widać w powyższych przykładach, „duck typing” jest wygodnym podejściem do typów. Pisząc programy w Ruby, które używają tego podejścia należy jedynie pamiętać, że obiekt jest klasyfikowany przez to, co może robić, a nie czym jest jego klasa. Przykład 52 pokazuje możliwość wykorzystania przez programistów Java lub C# typów w Ruby.

Przykład 52:

```
def create_url(server,path)
      raise TypeError.new('Server expected') if
      !server.kind_of?(Server)
      raise TypeError.new('String expected') if
      !path.kind_of?(String)
      url = 'http://' << server.host << '/' << path
       return url
end
```

Źródło: Opracowanie własne

Nie trzeba sprawdzać typu argumentów wywołania metody. Jeżeli obsługują metodę << lub **host** metoda będzie działać poprawnie. Jeżeli nie, interpreter i tak zwróci wyjątek.

Przykład 53 prezentuje wykorzystanie ‘duck typing’, dzięki czemu metoda staje się o wiele bardziej elastyczna i czytelna.

Przykład 53:

```
def create_url(server,path)
      return 'http://' << server.host << '/' << path
end
```

Źródło: Opracowanie własne

Interpreter oraz standardowa biblioteka języka Ruby używają specjalnych protokołów do wykonywania operacji, które w innych językach wymagałyby użycia typów danych. Protokoły te pozwalają konwertować obiekty do innych klas na trzy sposoby.

Pierwszym sposobem są metody **to_s** oraz **to_i,** które konwertują swojego odbiorcę do ciągu znaków lub liczby całkowitej. Nie są one ścisłe, co oznacza, że jeżeli obiekt może być „przyzwoicie” przedstawiony jako ciąg znaków, będzie prawdopodobnie posiadał metodę to_s.

Drugą formą konwersji są metody **to_str** oraz **to_int**. W odróżnieniu od pierwszego sposobu są one ścisłe. Implementowane są tylko wtedy, jeżeli obiekt może być naturalnie użyty jako ciąg znaków lub liczba całkowita. Przykład 54 prezentuje klasę File oraz metodę otwierania pliku.

Przykład 54:

```
class File

      def File.new(file,*args)
            if file.respond_to?(:to_int)
                  IO.new(file.to_int,*args)
            else
                  name = file.to_str
            end
      end
end
```

Źródło: [4]

Ruby nie sprawdza wprost, czy pierwszy parametr metody File.new jest obiektem klasy Fixnum lub String. Zamiast tego, pozwala obiektowi przedstawić się jako ciąg znaków lub liczba całkowita.

W standardową bibliotekę Ruby wbudowana została niewielka ilość ścisłych metod konwersji obiektów. Przedstawione są one w tabeli 2.

Metoda	Rezultat	Metoda	Rezultat
to_ary	Array	to_proc	Proc
to_hash	Hash	to_str	String
to_int	Integer	to_sym	Symbol
to_io	IO		

Tabela 2. Metody konwersji obiektów w języku Ruby
Źródło: Opracowanie własne

Trzeci sposób konwersji obiektów w języku Ruby odnosi się tylko do wartości numerycznych. Polega on na wymuszeniu typu. Kiedy piszemy „5+10”, interpreter wie, że ma wykonać operację + na obiekcie 5 (Fixnum) przekazując jej obiekt 10 (również Fixnum). Jednak, kiedy napiszemy „5+10.2”, ta sama metoda + otrzyma instancję klasy Float (10.2).

Protokół wymuszenia typu bazuje na metodzie coerce. Jej podstawowa wersja jest bardzo prosta. Potrzebuje ona dwóch numerycznych argumentów, z których pierwszy jest obiektem, na którym jest wykonywana, a drugi jest jej parametrem. Metoda coerce zwraca dwu-elementową tablicę zawierającą reprezentację obydwu liczb. Kolejność w tablicy jest odwrotna, najpierw podany jest parametr metody, następnie jej odbiorca. Metoda coerce gwarantuje, że oba te obiekty będą tej samej klasy i że mogą być na nich wykonywane operacje arytmetyczne. Przykład 55 prezentuje sposób użycia metody coerce, oraz jej przykładowe rezultaty.

Przykład 55:

```
3.coerce(4)          # => [4, 3]
4.coerce(6.3)        # => [6.3, 4.0]
(3.2).coerce(10.3)   # => [10.3, 3.2]
(2.1).coerce(1)      # => [1.0, 2.1]
```

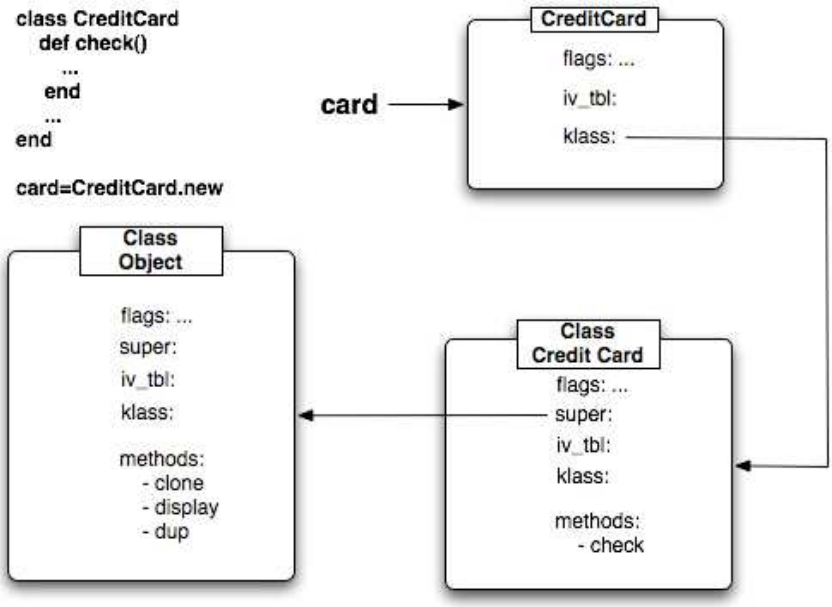
Źródło: Opracowanie własne

Wracając do metody „+”, okazuje się, że odbiorca wywołuje metodę coerce, aby wygenerować taką tablicę. Ta technika, nazywana „podwójną wysyłką” (double dispatch) pozwala metodzie zmienić swoje zachowanie nie tylko na podstawie klasy obiektu, który jest odbiorcą, ale również na podstawie swoich parametrów.

Duck typing jest kontrowersyjnym podejściem do typów w programowaniu. Ma ono swoich zagorzałych zwolenników jak i przeciwników. Należy pamiętać, że nie jest on zbiorem reguł, ale stylem programowania.

4.1.3 Klasy i Obiekty

Obiekt w języku Ruby składa się z trzech komponentów. Jest to zbiór flag, zbiór zmiennych instancji oraz przypisana klasa. Klasa jest obiektem klasy Class. Wszystkie metody wywoływane w Ruby określają odbiorcę, którym domyślnie jest **self** – aktualny obiekt (odpowiednik this w Javie). Ruby wyszukuje metody do wywołania na liście dostępnych metod dla klasy odbiorcy. Jeżeli jej tam nie znajdzie przeszukuje wszystkie załączone moduły, następnie klasę nadrzędną, jej moduły, jej klasę nadrzędną itd. Jeżeli metoda nie zostanie odnaleziona Ruby wykonuje metodę method_missing na odbiorcy. Rysunek 9 przedstawia obiekt **card**, jego klasę **CreditCard** oraz superklasę **Object**.

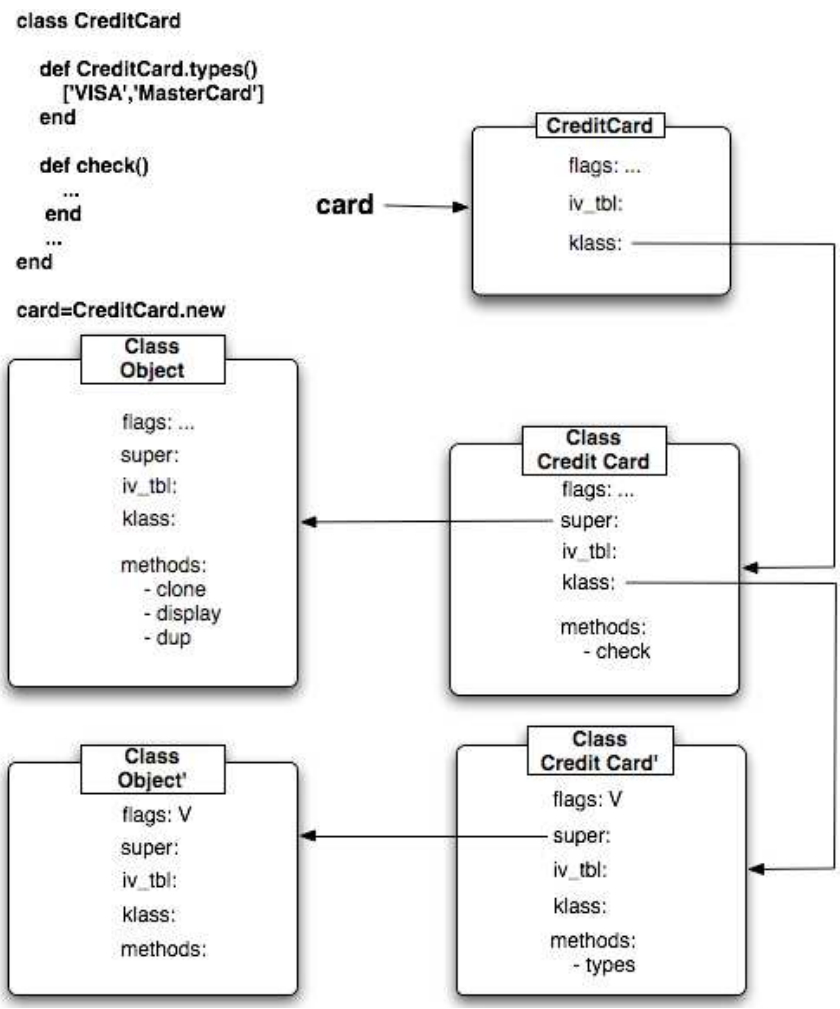


Rysunek 9. Obiekt card, klasa CreditCard oraz superklasa Object w języku Ruby.
Źródło: Opracowanie własne

Jeżeli wywołana zostanie metoda **card.check(),** Ruby odwoła się do odbiorcy metody **card** i podąży za referencją **klass** do klasy **CreditCard**. Tam metoda check zostanie wyszukana oraz wykonana. Jeżeli wywołana zostanie metoda **card.display()**, Ruby będzie jej szukał w ten sam sposób co w przypadku metody card.check(). Nie znajdzie jej jednak w klasie CreditCard. Będzie jej szukał w klasie nadrzędnej **Object**. Jeśli ją tam znajdzie, metoda display zostanie wykonana.

Metaklasy

Rysunek 10 prezentuje sytuację, w której wywoływana metoda jest metodą klasy, a nie obiektu.



Rysunek 10. Klasy wirtualne w języku Ruby.
Źródło: Opracowanie własne

Odbiorcą metody **types()** jest w tym przypadku obiekt-klasa **CreditCard**. Twórcy języka Ruby, chcieli być konsekwentni w swojej pracy i postanowili taką metodę klasy umieścić w innej klasie, dołączonej do **CreditCard** wskaźnikiem **klass**. Ta nowa klasa będzie posiadać wszystkie metody klasy CreditCard i nazywana jest **metaklasą**. Na rysunku 10, metaklasy oznaczone są znakiem apostrofu przy nazwie. Ponieważ CreditCard jest podklasą klasy **Object**, jej metaklasa CreditCard' jest podklasą metaklasy **Object'**.

Kiedy Ruby wykonuje metodę **CreditCard.types()** przeprowadzany jest taki sam proces wyszukiwania metody w klasach, jak w przypadku poprzedniej metody check(). Interpreter zgłasza się do odbiorcy - klasy CreditCard, następnie podąża referencją klas do metaklasy CreditCard'. Tam metoda zostaje odnaleziona i wykonana.

Na rysunku 10, metaklasy zostały oznaczone flagą V. Są one automatycznie tworzone przez Ruby i oznaczone jako **klasy wirtualne (virtual classes).** Są one niewidoczne dla programisty oraz nie można utworzyć ich instancji.

Ruby pozwala tworzyć klasy dopasowane do konkretnych instancji. Przykład 56 pokazuje sposób asocjacji anonimowej klasy z obiektem klasy Spring.

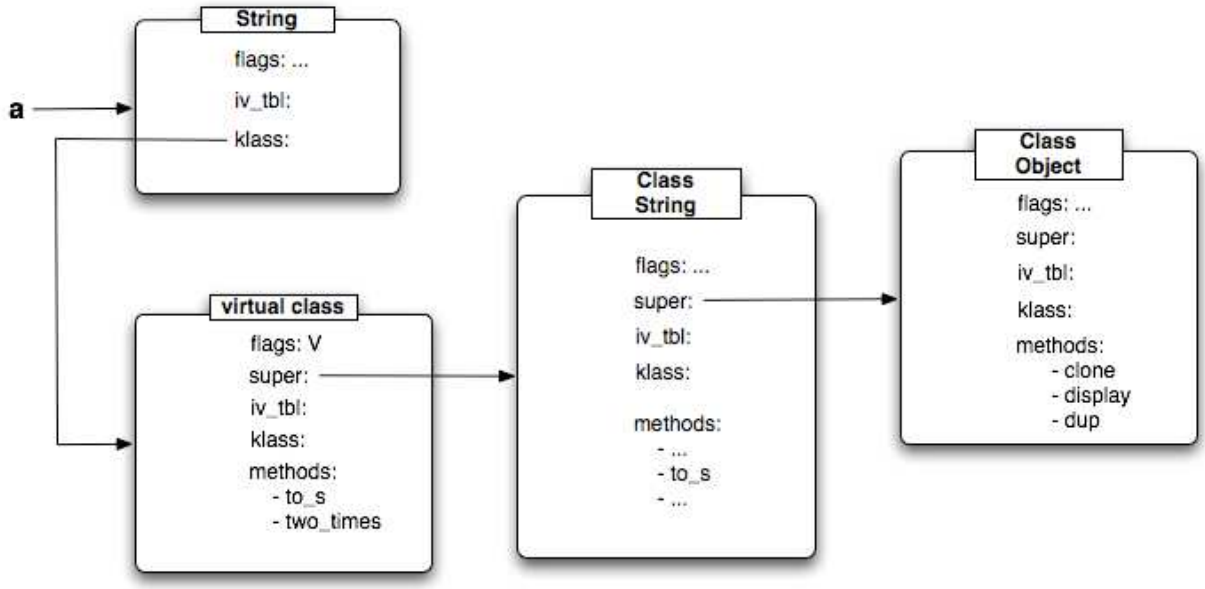
Przykład 56:

```
a = "Ruby"
b = a.dup
class << a
```

```
        def to_s
              "The value is '#{self}'"
        end

        def two_times
              self+self
        end
  end

  a.to_s      # => "The value is 'Ruby'"
  a.two_times # => "hellohello"
  b.to_s      # => "hello"
```

Źródło: [4]

Przykład pokazuje użycie notacji **„<< obiekt”**, która oznacza, że zostanie zbudowana nowa klasa wyłącznie dla obiektu **obiekt.** Wirtualna klasa zostanie utworzona jako zwykła klasa. Klasa String staje się w tym przypadku klasą nadrzędną dla nowej klasy. Struktura obiektów i klas dla tego przykładu zaprezentowana jest na rysunku 11.



Rysunek 11. Klasy wirtualne dla obiektu w języku Ruby.
Źródło: [4]

Moduły

Jeśli klasa posiada dołączony moduł, metody instancji dla modułu stają się metodami instancji klasy. Odpowiada to sytuacji, w której moduł byłby klasą nadrzędną dla danej klasy. Gdy moduł jest dołączany do klasy (poprzez polecenie „include”), Ruby tworzy klasę będącą anonimowym proxy, która połączona jest z modułem i stanowi bezpośrednią klasę nadrzędną dla danej klasy.

Anonimowe proxy zawiera referencje do zmiennych instancji oraz metod dostępnych w module. Dzięki temu, moduł może zostać dołączony do wielu różnych klas i może pojawiać się w wielu różnych łańcuchach dziedziczenia. Dzięki klasie proxy, nadal odnosimy się do jednego, tego samego modułu. Zmiana metody w module spowoduje zmianę we wszystkich klasach, do których ten moduł jest dołączony. Sposób włączania modułów do klas zaprezentowany jest w przykładzie 57.

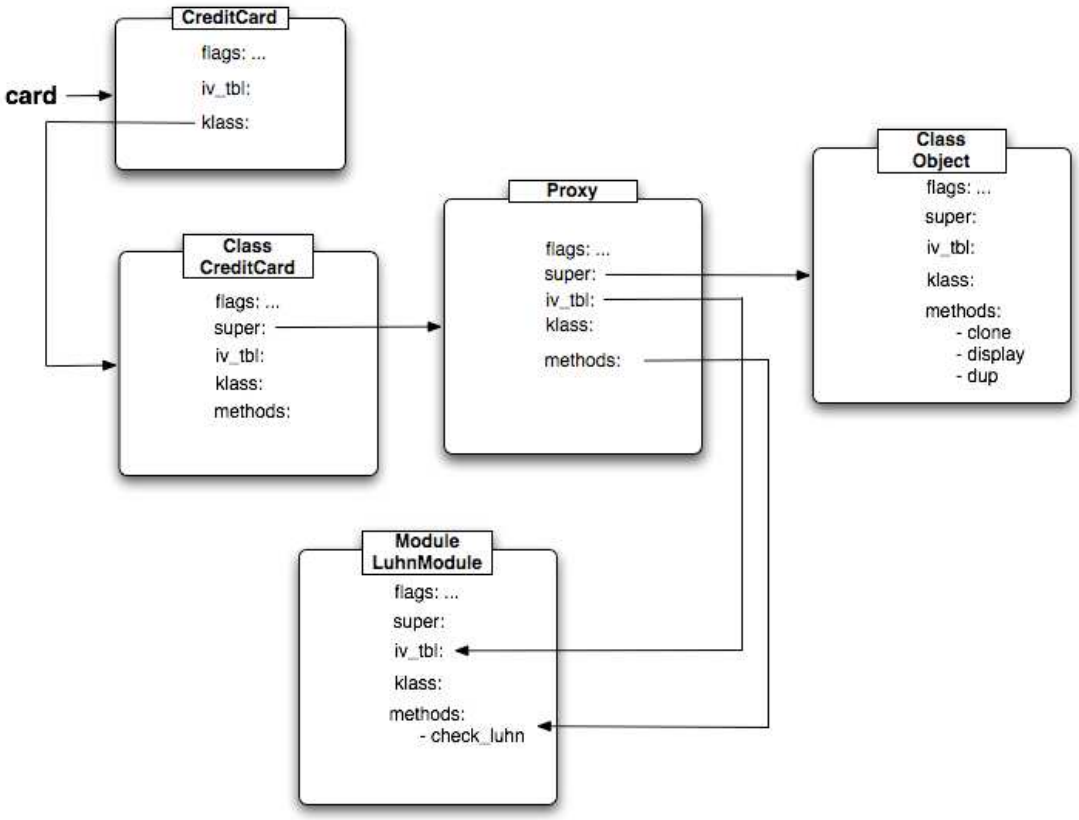
Przykład 57:

```
module LuhnModule
	def check_luhn
		self.number = self.number.gsub(/D/, '')
		cardLength = self.number.length
		parity = cardLength % 2

		sum = 0
		for i in 0...cardLength
			digit = self.number[i] - 48
			if i % 2 == parity
				digit = digit * 2
			end

			if digit > 9
				digit = digit - 9
			end

			sum = sum + digit
		end
		return (sum % 10) == 0
	end
end

class CreditCard
	include LuhnModule
	attr_accessor :number
end

card=CreditCard.new
card.number = '4012888888881881'
card.check_luhn # => true
```

Źródło: Opracowanie własne

Moduł **LuhnModule** posiada jedną metodę **check_luhn**. Jest to metoda, która algorytmicznie sprawdza poprawność numeru karty kredytowej. Klasa **CreditCard** ma dołączony moduł LuhnModule przy pomocy polecenia **„include LuhnModule”**.

Metoda instancji modułu staje się tym samym metodą instancji klasy. Jak widać, obiekt **card** posiada metodę „check_luhn”, dzięki której możliwe jest sprawdzenie poprawności numeru karty kredytowej, który jest jej zmienną instancji **(self.number)**. Metoda check_luhn operuje na zmiennej instancji bez potrzeby deklarowania jej w module (zakłada się, że klasa docelowa będzie ją posiadać). Struktura obiektów dla tego przypadku przedstawiona została na rysunku 12.



Rysunek 12. Klasa proxy w języku Ruby.
Źródło: Opracowanie własne

Rozszerzanie obiektów

Tak jak można definiować anonimową klasę dla obiektu przy użyciu notacji **class << obiekt**, tak samo można dołączać moduł do istniejącego już obiektu przy użyciu metody **extend**. Przykład 58 prezentuje sposób rozszerzania obiektów o moduł.

Przykład 58:

```
module LuhnModule
      def check_luhn
            ...
      end
end

class CreditCard
      attr_accessor :number
end

card=CreditCard.new
card.number = '4012888888881881'
card.extend LuhnModule
card.check_luhn # => true
```

Źródło: Opracowanie własne

W tym przypadku, moduł LuhnModule dołączony został do obiektu a nie klasy (jak w przykładzie 56). Metoda check_luhn dla obiektu jest dostępna dopiero wtedy, gdy zostanie on rozszerzony o interesujący nas moduł.

Metoda **extend**, może być użyta na dwa sposoby. Jeżeli zostanie użyta w definicji klasy, metody modułu staną się metodami klasy (nie instancji). Dzieje się tak, ponieważ umieszczenie metody **extend** w definicji klasy jest równoznaczne z wywołaniem **self.extend**, gdzie self jest daną klasą. Przykład 59 prezentuje sposób użycia metody extend w definicji klasy.

Przykład 59:

```
module Card
      def types
            %w<Visa MasterCard>
      end
end

class CreditCard
      extend Card
      include Card
end

CreditCard.types # => ["Visa","MasterCard"]

card=CreditCard.new
card.types       # => ["Visa","MasterCard"]
```

Źródło: Opracowanie własne

Klasa **CreditCard** jest rozszerzona o (**extend**) oraz posiada dołączony (**include**) moduł **Card**. **Extend** sprawia, że metody modułu Card staną się metodami klasy CreditCard. Natomiast **include** sprawia, że metody modułu Card staną się metodami instancji klasy CreditCard.

4.1.4 Metaprogramowanie

Metaprogramowanie to programowanie na poziomie klas, czyli obiektów klasy Class.

Wykonywanie kodu w kontekście odbiory

Technika wykonywania kodu w kontekście odbiorcy polega na dostępie do definicji klasy lub modułu w czasie wykonania. Jest to możliwe przy użyciu metod class_eval oraz module_eval. Metody te pozwalają dodać kod do istniejących już klas lub modułów.

Przykład 60 prezentuje użycie metody class_eval, której parametrem jest blok lub łańcuch znaków. Metoda ta powoduje dołączenie zadanego bloku (lub kodu) do klasy, w kontekście, której została wywołana.

Przykład 60:

```
class Class
      def attr_access(*args)
            args.each do |attr|
                  class_eval %Q{
                        def #{attr}
                              @#{attr}
                        end

                        def #{attr}=(value)
                              @#{attr} = value
                        end
                  }
            end
      end
```

```
end

class CreditCard
      attr_access :number,:holder
end

card = CreditCard.new
card.number='4000000000000010'
card.number   # => '4000000000000010'
```

Źródło: Opracowanie własne

Przykład ten pokazuje sposób dynamicznego tworzenia metod w klasie. Klasa **CreditCard** wykonuje jedynie metodę **attr_access** przekazując do niej dwa symbole. Metoda attr_access iteruje po tych symbolach i dla każdego z nich, przy użyciu metody **class_eval**, dodaje metody dostępowe (setter i getter) dla danej klasy. W podobny sposób można definiować metody dostępowe to zmiennych klasy (oznaczanych przez @@). Przykład 61 prezentuje sposób użycia class_eval dla tego przypadku.

Przykład 61:

```
class Class
      def cattr_access(*args)

            args.each do |attr|
                  class_eval %Q{
                        def self.#{attr}
                              @@#{attr}
                        end
                        def self.#{attr}=(value)
                              @@#{attr} = value
                        end
                  }
            end
      end
end
class CreditCard
      cattr_access :types
end
CreditCard.types=['Visa','MasterCard']
CreditCard.types  # = ['Visa','MasterCard']
```

Źródło: Opracowanie własne

Tym razem klasa **CreditCard** wykonuje metodę cattr_access przyjmując jeden symbol jako argument. Metoda **class_eval** tworzy w tym przypadku metody dostępowe dla zmiennych klasy, przez co możliwe jest operowanie na nich z zewnątrz klasy.

Metoda **class_eval** wymaga, aby kod ukryty był w łańcuchu znaków. Jedną z alternatyw jest stosowanie bloków kodu. Drugą jest wykorzystanie metody **define_method**, która powoduje zdefiniowanie metody w kontekście klasy, na rzecz, której została wykonana. To, co różni ją od słowa kluczowego def, to określanie nazwy metody poprzez symbol lub łańcuch znaków. Przykład 62 prezentuje sposób użycia metody define_method, dla zastąpienia class_eval z przykładu 60.

Przykład 62:

```
class Class
      def attr_access(*args)
            args.each do |attr|
                  define_method attr do
```

```
                        @attr
                    end

                    define_method "#{attr}=" do |value|
                        @attr=value
                    end
                end
            end
        end

        class CreditCard
            attr_access :number,:holder
        end

        card = CreditCard.new
        card.number='4000000000000010'
        card.number   # => '4000000000000010'
        card.holder='Slawomir Zabkiewicz'
        card.holder  # => 'Slawomir Zabkiewicz'
```

Źródło: Opracowanie własne

W przypadku użycia define_method, jej argumenty pojawiają się jako parametry bloku (w tym przypadku „value”), co czyni jej zapis bardziej skomplikowanym. Pomimo to define_method jest bardziej naturalnym sposobem definiowania metod w czasie wykonania programu.

Definiowanie metod klasowych

Jak już wspomniano, klasy w Ruby są również obiektami, a metody klas są metodami singletonowymi (metaklas) tych obiektów. Dodawanie metod klas odbywa się na tej samej zasadzie, co w przypadku użycia metod class_eval lub define_method. Jedyną różnicą jest to, że metody te wykonywane są nie na rzecz modyfikowanej klasy, lecz na rzecz jej metaklasy. Przykład 62 prezentuje sposób pobrania metaklasy dla danej klasy.

Przykład 62:

```
class << self; self;
end
```

Źródło: Opracowanie własne

Self reprezentuje w tym przypadku dwa różne obiekty. Pierwsze oznacza obiekt klasy, do której singletonu rządany jest dostęp. Drugie natomiast odnosi się do tego konkretnego singletonu.

W przykładzie 61 zaprezentowana została metoda cattr_access, która przy użyciu metody class_eval definiowała metody dostępowe dla zadanych argumentów. Tak naprawdę, użycie w tych metodach słowa kluczowego self oraz tworzenie tych metod spowodowało dodanie ich do odpowiedniej metaklasy. Przykład 64 prezentuje taką samą sytuację, ale z użyciem bezpośrednio metaklasy.

Przykład 64:

```
class Class
    def cattr_access(*args)
        args.each do |attr|
            (class << self; self; end).class_eval %Q{
                def #{attr}
                    @#{attr}
                end
                def #{attr}=(value)
```

```
                        @#{attr} = value
                    end
                }
            end
        end
    end
    class CreditCard
        cattr_access :types
    end
    CreditCard.types=['Visa','MasterCard']
    CreditCard.types  # = ['Visa','MasterCard']
```

Źródło: Opracowanie własne

Tym razem metoda **class_eval** wykonywana jest bezpośrednio na metaklasie. Ten sam przykład można zapisać dużo krócej, wykorzystując wbudowane w Ruby metody attr_reader oraz attr_writer. Rozwiązanie to działa, ponieważ wywołania te realizowane są w kontekście klas singletonowych poszczególnych klas. Oznacza to, że odpowiednie metody dostępowe, stają się metodami klasowymi tych klas. Przykład 65 prezentuje użycie metod attr_writer oraz attr_reader dla metaklasy.

Przykład 65:

```
    class Class
        def cattr_access(*args)
            args.each do |attr|
                (class << self; self; end).class_eval do
                    attr_reader attr
                    attr_writer attr
                end
            end
        end
    end
```

Źródło: Opracowanie własne

Z przedstawionych przykładów wynika, że rozszerzanie języka Ruby nie jest trudne. Nie należy jednak przesadzać z wprowadzaniem wielu meta-poziomów, gdyż kod może stać się bardzo mało czytelny i trudny w utrzymaniu.

Modyfikowanie atrybutów instancyjnych

W Ruby występują metody służące do manipulowania zmiennymi instancji. Są to metody **instance_variable_get** oraz **instance_variable_set**. Metoda get pozwala odczytać, a metoda set pozwala zmienić wartość zmiennej o nazwie z parametru. Przykład 66 prezentuje sposób użycia metod instance_variable_get oraz instance_variable_set.

Przykład 66:

```
    class Class
        def meta
            (class << self; self; end)
        end

        def cattr_access(*args)
            args.each do |attr|
                meta.class_eval do
                    define_method attr do
                        instance_variable_get("@#{attr}")
                    end

                    define_method "#{attr}=" do |val|
```

```
                                instance_variable_set("@#{attr}",val)
                            end
                        end
                    end
                end
            end

            class CreditCard
                cattr_access :types
            end

            CreditCard.types=['Visa','MasterCard']
            CreditCard.types  # = ['Visa','MasterCard']
```

Źródło: Opracowanie własne

W tym przypadku dodatkowo wydzielona została metoda **meta** zwracająca metaklasę danej klasy. To na niej wykonywana jest metoda **class_eval**, której blok stanowią dwa wywołania metody **define_method.** Operowanie na metaklasie pozwala na utworzenie jej metod instancji, które z kolei stają się metodami klas dla klasy docelowej.

4.1.5 Podsumowanie

Ruby jest bardzo dynamicznym językiem programowania. Dynamika ta sprawiła, że metaprogramowanie zyskuje szczególne znaczenie w kontekście tego języka. Implementacja wzorca projektowego Dependency Injection w Ruby nie mogłaby się obejść bez konstrukcji pozwalających na ingerowanie w klasy i obiekty w czasie wykonania (runtime).

Jest to istotne, gdyż kontener Inversion of Control musi automatycznie utworzyć instancje komponentów zadeklarowanych przez programistę. W proponowanym rozwiązaniu zostaną szeroko wykorzystane techniki wykonywania kodu w kontekście odbiorcy oraz modyfikowania atrybutów instancyjnych.

5. Implementacja wzorca Dependency Injection w Ruby

Poniższy rozdział prezentuje kompletną implementację wzorca Dependency Injection w języku Ruby. Opisana jest przyjęta koncepcja kontenera Inversion of Control oraz sposób wstrzykiwania zależności do komponentów. Dodatkowo wprowadzone zostały pojęcia zakresów (scopes) zależności oraz opisane sposoby ich użycia.

Kolejną część rozdziału stanowi przykład praktycznego użycia zaprezentowanej biblioteki. Jest nim prosty serwer XML-RPC, którego zdalne obiekty tworzone są przez prezentowaną bibliotekę Dependency Injection. Aby dobrze zrozumieć przydatność wstrzykiwania zależności, zostanie opisana specyfikacja XML-RPC, biblioteka LibXML służąca do parsowania dokumentów XML oraz sposób instalacji oraz użycia.

5.1. Biblioteka dla języka Ruby

Do zbudowania biblioteki użyty został język Ruby 1.9. Ta wersja języka zawiera wiele zmian w stosunku do wersji wcześniejszych. Główną zaletą jest jej szybkość, którą osiągnięto poprzez zamianę dotychczasowego MRI (Matz Ruby Intepreter) na YARV(Yet Another Ruby Vm), czyli maszynę wirtualną. Nowa implementacja jest około 3-4 razy szybsza i posiada optymalizację operacji na liczbach oraz mniejsze zużycie pamięci.

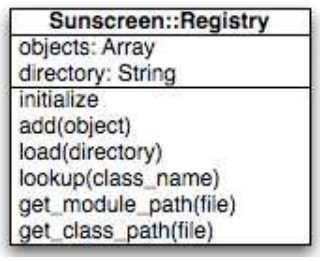
Sam język również uległ wielu zmianom, a niektóre budzą obawy czy były potrzebne, ale widoczne jest zdecydowanie twórcy języka (Yukihiro Matsumoto) w ich wprowadzaniu. Ponieważ wszystkie klasy języka są otwarte, nie ma problemu z dostosowywaniem istniejących aplikacji do nowej wersji języka.

Prezentowane rozwiązanie składa się z trzech podstawowych elementów:

- Kontenera Inversion of Control
- Mechanizmu wstrzykiwania zależności (wraz z zakresami)
- Mechanizmu przechwytywania wywołań w metodach

5.1.1 Kontener Inversion of Control

Zasada działania kontenerów Inversion of Control została opisana w rozdziale drugim. Na podstawie konfiguracji programisty obiekty powinny zostać automatycznie utworzone przez system. W proponowanym rozwiązaniu kontener obiektów jest instancją klasy **Registry**, która w atrybucie **objects** przetrzymuje utworzone obiekty. Rysunek 13 przedstawia klasę Registry:



Rysunek 13. Diagram klasy Sunscreen::Register.
Źródło: Opracowanie własne

Konstruktor klasy Registry (metoda **initialize**) jest bezparametrowy. Jego zadaniem jest zainicjowanie tablicy obiektów. Aby wypełnić tablicę obiektów zgodnie z konfiguracją programisty należy wykonać metodę load, której parametrem jest ścieżka do katalogu.

Jest to pierwsza różnica w podejściu do kontenera obiektów, jaki prezentowały implementacje Dependency Injection opisane w rozdziale 3. Proponowane rozwiązanie zakłada budowanie grafu obiektów na podstawie lokalizacji plików klas. Podejście to zostało pierwszy raz użyte we frameworku MVC Ruby on Rails. Jest ono zgodne z założeniami programowania w Javie mówiącymi, że nazwa pliku odpowiada nazwie klasy, której definicja się w nim znajduje. Założenie to jest wygodne w systemie, w którym tworzenie pewnych bytów musi odbyć się automatycznie. Dodatkowo zakresy nazw odpowiadają w prezentowanym rozwiązaniu strukturze katalogów.

Tworzenie nowej instancji kontenera Registry nie spowoduje utworzenia grafu obiektów. Aby tak się stało należy wykonać na obiekcie klasy Registry metodę **load**, której parametrem jest bezwzględna ścieżka do katalogu z obiektami, które mają być zarządzane przez kontener. Przykład 67 prezentuje metodę **load**, która tworzy instancje obiektów kontenera.

Przykład 67:

```
def load(directory)
@directory=directory if @directory.nil?
      Dir.glob(directory+'/*').each do |file|
            if file.split('.').last=='rb'
                  require file
                  file.gsub!(@directory,"")
                  module_path=get_module_path(file)

                  class_path=get_class_path(file)
                  module_path != '' ?
                  class_name=module_path+'::'+class_path :
                  class_name=class_path

                  add(eval("#{class_name}.new"))

            elsif File.directory?(file)
                  load(directory+'/'+file.split('/').last)
            end
      end
end
```

Źródło: Opracowanie własne

Metoda Dir.glob zwraca nazwy plików znalezione dla zadanego wzorca. Możliwe wartości wzorca przedstawione zostały w tabeli 3.

Wzorzec	**Znaczenie**
*	Odpowiada każdemu plikowi. Może być ograniczony innymi wartościami metody glob. Np. c* będzie odpowiadać wszystkim plikom zaczynającym się od litery c
**	Odpowiada katalogom (rekursywnie)
?	Odpowiada każdemu możliwemu znakowi. Odpowiedniek /.{1}/ w regexp
[set]	Odpowiada każdemu znakowi w tablicy set. Odpowiednik np. [a-z] w regexp
{p,q}	Odpowiada znakowi p lub znakowi q.

Tabela 3. Wzorce dla metody Dir.glob.
Źródło: Opracowanie własne

W przypadku przedstawionej w przykładzie 65 metody **load** konetenera obiektów pełnym wzorcem dla metody Dir.glob jest **directory+'/*'** co oznacza, że wyszukane zostaną wszystkie pliki oraz katalogi znajdujące się w katalogu, którego ścieżka została podana w parametrze wykonania metody. Metoda load jest rekurencyjna, dla napotkanego katalogu przeszukuje go w głąb. Odnalezione pliki są filtrowane w poszukiwaniu plików o rozszerzeniu .rb. Służy do tego metoda file.split('.').last=='rb', która rozkłada napotkany plik na części oddzielone kropką, a następnie sprawdza czy ostatni element jest zgodny z poszukiwanym rozszerzeniem. Jeżeli warunek jest prawdą następuje załączenie pliku do środowiska systemu przy pomocy metody require file.

Metoda **require** to odpowiednik **import** w Java. Powoduje, że klasy oraz metody z zewnętrznego pliku będą dostępne do użycia w projekcie, który wykonuje konkretną metodą require. Metoda require powoduje załadowanie pliku docelowego tylko raz. Ponowne wykonanie tego samego polecenia zwróci **false** a plik nie zostanie przeładowany. Aby móc przeładować istniejący już w systemie plik należy użyć metody Ruby **load.**

Po załadowaniu pliku do systemu następuje skrócenie jego nazwy przez usunięcie ścieżki bezwzględnej do katalogu. Jest to możliwe, ponieważ w metodzie load zawsze jest dostępna nazwa przeszukiwanego katalogu. Czynność ta wykonana jest z użyciem metody **file.gsub!(@directory,"").**

Gsub to rodzina metod, które zwracają łańcuch znaków, w którym wszystkie wystąpienia wzorca (podanego jako pierwszy argument) zostały zastąpione zamiennikiem (podanym jako drugi argument). Wzorzec to zazwyczaj Regexp. Możliwe jest również podanie łańcucha znaków, ale wówczas żadne metaznaczniki wyrażeń regularnych nie zostaną wzięte pod uwagę przez interepreter języka Ruby.

Tak przygotowana nazwa pliku jest gotowa do użycia przez kontener Registry. Należy teraz utworzyć obiekt klasy, której definicja zawarta jest w pliku. Wspomniano już wcześniej, że nazwy klas odpowiadają nazwom plików, w których się znajdują. Zakresy nazw z kolei odpowiadają nazwom katalogów. Mając lokalizację pliku (wraz z jego nazwą) w bardzo prosty sposób tworzona jest pełna ścieżka zakresów nazw oraz nazwa klasy. Służą do tego dwie metody prywatne klasy Registry: **get_module_path** oraz **get_class_path**. Przykład 68 przedstawia obie wymienione metody.

Przykład 68:

```
private
def get_module_path(file)
      file.split('/')[1..-2].collect{|p| p.camelize}.join('::')
end

def get_class_path(file)
      file.split('/').last.gsub!('.rb','').camelize
end
```

Źródło: Opracowanie własne

Jak widać pobranie nazw modułów (zakresów) polega na rozłożeniu ścieżki względnej do pliku (dzieląc po znaku **'/'**) oraz zamianie pierwszej litery nazw katalogów na dużą literę. Nazwy te są potem łączone (przy pomocy metody **join**) podwójnym znakiem dwukropka **::**.

Pobranie nazwy klasy odbywa się analogicznie. Ścieżka względna do pliku jest dzielona, ale pod uwagę brany jest tylko ostatni element. Usuwane jest rozszerzenie pliku, a pierwsza litera nazwy jest zamieniana na dużą. Łącząc wynik działania obu tych metod

otrzymywana jest pełna nazwa klasy, której instancję należy utworzyć a następnie umieścić na liście obiektów kontenera.

Jeżeli wynik działania metody **get_modue_path** jest pustym ciągiem znaków oznacza to, że wyszukana klasa znajduje się na pierwszym poziomie zagnieżdżenia katalogów. Wówczas nazwa klasy jest tworzona jedynie z wyniku działania metody **get_class_path**.

Tabela 4 przedstawia przykładowe katalogi przeszukiwane przez kontener oraz odpowiadające im pełne nazwy klas (wraz z zakresami). Katalog wyjściowy dla kontenera to **trunk/servants**.

Ścieżka do pliku	**Nazwa klasy**
trunk/servants/**vm_servant.rb**	VmServant
trunk/servants/**host_servant.rb**	HostServant
trunk/servants/network_servant.rb	NetworkServant
trunk/servants/validations/host_validation.rb	Validations::HostValidation
trunk/servants/validations/network_validation.rb	Validations::NetworkValidation
trunk/servants/validations/vm_validation.rb	Validations::VmValidation

Tabela 4. Ścieżki oraz nazwy klas obiektów kontenera Inversion of Control.
Źródło: Opracowanie własne

Po ustaleniu pełnej nazwy klasy, kontener tworzy jej instancję przy pomocy metody eval. Metoda eval oblicza wartość ciągu znaków, tak jakby był kodem programu w języku Ruby. Przykład 69 prezentuje użycie metody **eval** do utworzenia instancji obiektu klasy, której nazwa zawarta jest w zmiennej **class_name**.

Przykład 69:
```
eval("#{class_name}.new")
```

Źródło: Opracowanie własne

Wynikiem działania metody eval jest instancja danej klasy. Kontener dodaje ją do listy obiektów przy pomocy metody **add**, po czym przechodzi do pracy z kolejnym plikiem lub katalogiem.

Ostatecznie cały proces dodawania obiektów do kontenera kończy się w momencie osiągnięcia ostatniego katalogu. W efekcie pracy kontenera dla katalogów oraz plików podanych w tabeli 4, lista obiektów kontenera wyglądać będzie tak jak w przykładzie 70.

Przykład 70:
```
$registry.objects.each{|obj| obj.inspect}.join("\n")
# =>
#<HostServant:0x119b120>
#<NetworkServant:0x119ab80>
#<Validations::HostValidation:0x119a130>
#<Validations::NetworkValidation:0x11999d8>
#<Validations::VmValidation:0x119926c>
#<VmServant:0x11987e0>
```

Źródło: Opracowanie własne

Zostały utworzone instancje wszystkich klas, których pliki znajdowały się w podkatalogach katalogu pracy kontenera. Aby uniknąć tworzenia obiektów przez kontener, programista musi wydzielić osobny katalog w projekcie, w którym znajdować się będą pliki z definicjami klas dla kontenera. Biblioteka nie pozwala na oznaczanie klas jako niewidoczne dla kontenera. Do twórcy oprogramowania należy decyzja, które pliki zostaną użyte przez kontener.

Przykład 71 prezentuje sposób utworzenia nowego kontenera oraz przekazania mu katalogu startowego.

Przykład 71:

```
$registry=Sunscreen::Registry.new
path=File.expand_path(File.dirname(__FILE__) + '/servants')
$registry.load(path)
```

Źródło: Opracowanie własne

Zmienna zaczynająca się od znaku **$** oznacza w Ruby zmienną globalną. Pozwala to na uzyskanie dostępu do kontenera w każdym miejscu tworzonego programu bez konieczności przekazywania go w poszczególnych wywołaniach metod.

Parametr wywołania metody load jest wynikiem działania dwóch metod operujących na klasie **File**: **expand_path** oraz **dirname**. Metoda expand_path konwertuje ścieżkę względną do ścieżki bezwzględnej. Jej parametrem jest metoda dirname, która pobiera nazwę katalogu, w którym jest wykonywany program Ruby. Nazwa ta jest łączona z nazwą katalogu, który ma być zarządzany przez kontener. W ten sposób do metody **load** kontenera zostanie przekazana bezwzględna ścieżka do katalogu **servants** znajdującego się w katalogu głównym projektu (a przynajmniej pliku wykonywanego). Utworzona w ten sposób zmienna globalna jest instancją kontenera, gotową do użycia w projekcie.

5.1.2 Wstrzykiwanie zależności

Proponowana w tej pracy implementacja wzorca Dependency Injection, oparta jest na wstrzykiwaniu zależności przez operacje dostępowe (settery). Nie jest to jednak typowe Setter Dependency Injection, jakie występowało w większości implementacji opisanych w rozdziale 3. Stosowane podejście jest bardzo podobne do znanych z Google Guice adnotacji. Adnotacja @Inject mówiła kontenerowi, które atrybuty obiektu powinny być zarządzane przez kontener. Następnie poprzez odpowiednią implementację interfejsu Module, kontener wiedział obiekty jakich klas ma ze sobą połączyć.

W opisywanym w tym rozdziale rozwiązaniu stosowane jest podejście analogiczne do adnotacji z Google Guice. Adnotacje zostały zastąpione metodami dostępowymi, natomiast konfiguracja zależności odbywa się poprzez odpowiedni skład argumentów wywołania tych metod. Są to: **attr_injected** oraz **attr_injected_new.** Ponieważ wszystkie klasy w Ruby są otwarte, metody te zostały umieszczone w klasie Object, z której dziedziczą wszystkie inne klasy. Metody te są bardzo podobne do rodziny metod attr_* wbudowanych w Ruby, które służą do tworzenia atrybutów klas (zmiennych instancji) oraz odpowiadających im metod dostępowych. Ponieważ wstrzykiwanie zależności również opiera się na tworzeniu atrybutów klas, służące do tego metody zostały nazwane w tej samej konwencji (z przedrostkiem attr_). Przykład 72 przedstawia proste użycie metody attr_injected w klasie.

Przykład 72:

```
class VmServant

      attr_injected :logger

      def initialize
            ...
      end
      def deploy
            ...
      end
end
```

```
vm_servant=VmServant.new
vm_servant.logger # => #<Logger:0x22808>
```

Źródło: Opracowanie własne

Jak widać metoda **attr_injected** odnosi się do klasy a nie do obiektu i znajduje się na początku jej definicji. Jej argumentami są symbole lub ciągi znaków oddzielone przecinkami. Duże znaczenie ma nazewnictwo argumentów wywołania metody **attr_injected**.

Już wcześniej wspomniano, że dynamika języka Ruby pozwala na uproszczenie pewnych zadań. W tym wypadku celem było uniknięcie dodatkowej konfiguracji zależności w osobnych bytach programu (plikach xml, modułach). W związku z tym opracowana została konwencja nazewnictwa argumentów oparta na zakresach nazw obiektów zarządzanych przez kontener Inversion Of Control.

W poprzednim podrozdziale opisany został sposób przeszukiwania przez kontener katalogu z plikami zawierającymi definicje klas. Zgodnie ze strukturą katalogów odpowiednie klasy były grupowane w moduły (od nazw katalogów). Tak samo przy wstrzykiwaniu zależności do komponentów należy zaznaczyć zakresy nazw obiektów, które chcemy łączyć ze sobą.

W przykładzie 72 argumentem wywołania metody attr_injected jest symbol :logger. Oznacza to, że na wierzchołku grafu obiektów (w głównym katalogu przeszukiwania) znajduje się plik z definicją klasy Logger. Zmiana liter w nazwach klas oparta jest o metody camelize oraz underscore klasy String przedstawione na przykładzie 73.

Przykład 73:

```
class String

      def camelize(first_letter = :upper)
            case first_letter
                  when :upper then
                        gsub(/\/(.?)/) { "::" + $1.upcase }
                        .gsub(/(^|_)(.)/) { $2.upcase }
                  when :lower then
                        self.first + camelize(self)[1..-1]
            end
      end

      def underscore()
            gsub(/::/, '/')
            .gsub(/([A-Z]+)([A-Z][a-z])/,'\1_\2')
            .gsub(/([a-z\d])([A-Z])/,'\1_\2')
            .tr("-", "_")
            .downcase
       end
end
```

Źródło: Opracowanie własne

W opisywanym przykładzie metoda camelize usuwa znaki podkreślenia _ oraz zamienia poszczególne części nazwy na rozpoczynające się dużą literą. Tabela 5 przedstawia przykładowe działanie metody camelize na ciągach znaków.

Ciąg znaków	Camelize
v	V
logger	Logger

vm_servant	VmServant
vm_servant_validation	VmServantValidation
vm_servant.validation	VmServant.validation

Tabela 5. Przykład działania metody camelize.
Źródło: Opracowanie własne

Metoda underscore jest podobna, z tym że, znacznik zmiany zakresu nazw **::** zamienia na /, oraz myślnik **–** na podkreślenie **_**. Dodatkowo oczywiście następuje konwersja dużych liter na małe.

Konwencja nazw przyjęta przy wstrzykiwaniu zależności w opisywanym rozwiązaniu zakłada, że argumenty wywołania metody attr_injected będą w postaci underscore, system następnie sam dokona konwersji do nazw klas i modułów. Tabela 6 przedstawia przykładowe argumenty wywołania metody attr_injected wraz z odpowiadającymi im nazwami klas oraz ścieżkami przeszukiwania dla kontenera Inversion of Control (przy założeniu katalogu startowego w servants).

Argumenty metody	**Nazwa klasy**	**Ścieżka kontenera**
:v	V	servants/v.rb
:logger	Logger	servants/logger.rb
:vm_servant	VmServant	servants/vm_servant.rb
„validations.vm_validation”	Validations::VmValidation	servants/validations/vm_validation.rb
„db.handlers.postgresql”	Db::Handlers::Postgresql	servants/db/handlers/postgresql.rb

Tabela 6. Przykładowe wywołania metody attr_injected
Źródło: Opracowanie własne

Jak widać, aby wstrzyknąć obiekt znajdujący się w module (zakres nazw) należy przesłać do metody attr_injected ciąg znaków a zakresy nazw oddzielone powinny być znakiem kropki. Przykład 74 przedstawia użycie metody attr_injected, której argumentem jest odpowiednio przygotowana nazwa klasy zależności.

Przykład 74:

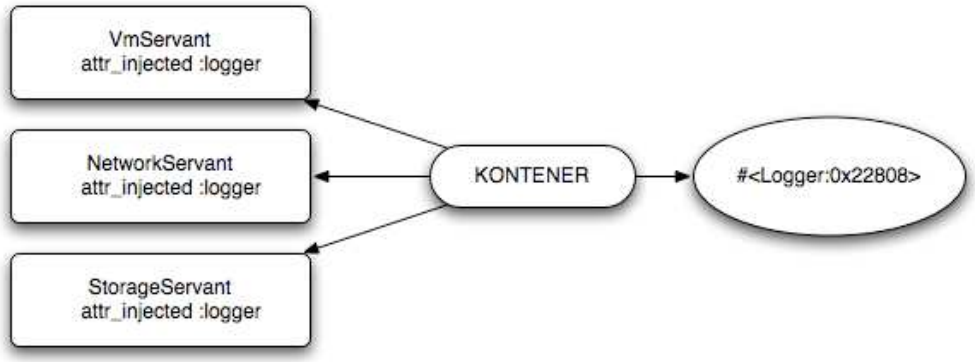
```
class VmServant
      attr_injected "validations.vm_validation"
      def initialize
       ...
      end
      def deploy
       ...
      end
end

vm=VmServant.new
vm.vm_validation # => #<Validations::VmValidation:0x21d18>
```

Źródło: Opracowanie własne

Zakresy zależności

Opisywana w tym rozdziale biblioteka dostarcza podstawowy model zakresów (scopes) dla obiektów zarządzanych przez kontener Inversion of Control. Zakres jest to kontekst, w ramach którego dany klucz (nazwa klasy) odnosi się do tej samej instancji obiektu. Innymi słowy jest to czas trwania obiektu. Zaletą zakresów jest to, że można je definiować w sposób deklaratywny. Opisywane rozwiązanie dostarcza dwa zakresy: singleton oraz prototype.

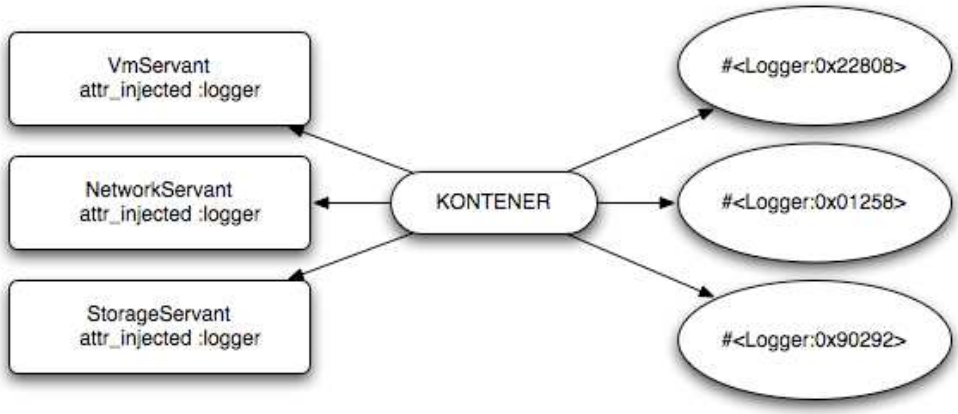
Zakres singleton oznacza, że tylko jedna instancja danej klasy jest tworzona przez kontener i używana dla zależnych od niej komponentów. W związku z tym deklarując zależność jako singleton w wielu różnych klasach, zagwarantowane jest, że do ich instancji zostanie wstrzyknięty ten sam obiekt. Rysunek 14 przedstawia wstrzykiwanie zależności klasy Logger do kilku obiektów różnych klas.



Rysunek 14. Przykład działania zakresu Singleton.
Źródło: Opracowanie własne

Jak widać każdy z trzech obiektów otrzyma tę samą instancję klasy **Logger**. Oznacza to, że kontener zarządza tylko jedną instancją danej klasy.

Czasem zachodzi potrzeba utworzenia oddzielnych instancji zależności dla różnych obiektów. Wówczas kontener Inversion of Control utworzy nowe obiekty i nie będzie nimi dalej zarządzał. Przypadek ten ilustruje rysunek 15.



Rysunek 15. Przykład działania zakresu Prototype.
Źródło: Opracowanie własne

Proponowany model zakresów jest bardzo prosty i ogranicza się tylko do stwierdzenia, czy zależność ma być pobrana z listy obiektów kontenera czy ma być stworzona nowa instancja danej klasy.

Wybór zakresu zależności odbywa się poprzez wywołanie innej metody, która spowoduje jej wstrzyknięcie. W przypadku zakresu singleton jest to opisywana już metoda **attr_injected**.

Aby wstrzyknąć zależność z zakresem prototype należy wywołać metodę **attr_injected_new**. Oznacza to, że zakresy zależności nie są definiowane w kontenerze,

ale w klasach, które z tych zależności korzystają. Jest to wygodne podejście gdyż pozwala zmieniać zakres nie ingerując w klasy zależności ani kontenera. Jedyne, co należy zrobić to zmiana nazwy metody służącej do wstrzykiwania.

Obie metody **attr_injected** oraz **attr_injected_new** stanowią rozszerzenie klasy **Object**, z której dziedziczą wszystkie inne klasy w Ruby. Dodanie tego rozszerzenia do projektu oznacza, że każda klasa w tym projekcie może z nich korzystać. Przykład 75 przedstawia obie te metody.

Przykład 75:

```
def attr_injected(*args)
      args.each do |arg|
            inject(arg,SCOPES['singleton'])
      end
end

def attr_injected_new(*args)
      args.each do |arg|
            inject(arg,SCOPES['prototype'])
      end
end
```

Źródło: Opracowanie własne

Obie te metody przyjmują zmienną ilość argumentów, dla których wywoływana jest metoda inject. Drugim argumentem tej metody jest znacznik numeryczny zakresu ze stałej SCOPES będącej tablicą asocjacyjną definiującą zakresy i ich znaczniki. Przykład 76 prezentuje metodę inject, która odpowiedzialna jest za wstrzyknięcie zależności do komponentu.

Przykład 76:

```
private
def inject(name,scope)
      name=name.to_s

      class_name=name.split('.').collect{
            |name| name.camelize
      }.join("::")
      var_name=name.split('.').last
      begin
            case scope
                  when 0 then
                        default=$registry.lookup(class_name)
                  when 1 then
                        default=$registry.create(class_name)
            end
      rescue NameError => e
            raise 'Object does not exists'
      end
      define_method(var_name) do
      unless instance_variable_defined?("@#{var_name.to_s}")
            instance_variable_set "@#{var_name.to_s}", default
      end
      instance_variable_get "@#{var_name.to_s}"
      end
  end
```

Źródło: Opracowanie własne

Odpowiednio podane nazwy zależności (jak opisano wyżej) dzielone są po znaku kropki. Złączenie wszystkich części podziału (odpowiednio zmienionych przez metodę camelize) podwójnym dwukropkiem **::** daje pełną nazwę klasy zależności w środowisku. Ostatnia część podziału stanowi nazwę nowej zmiennej w obiekcie.

Mając nazwę klasy można, w zależności od zakresu, pobrać lub utworzyć jej instancję. Zajmuje się tym utworzony na początku działania programu kontener **$registry**. Jeżeli zależność jest wstrzykiwana z zakresem **singleton**, wykonywna jest metoda **load**.

W przypadku użycia zakresu **prototype**, wykonywana jest metoda **create**. Obie te metody przyjmują jeden argument, którym jest nazwa poszukiwanej klasy.

Jeżeli kontener nie odnajdzie pasującego obiektu (w przypadku load) lub klasy (w przypadku create) zgłoszony zostanie wyjątek **NameError**. Następnie wykonywana jest metoda **define_method** (opisana w rozdziale dotyczącym metaprogramowania w języku Ruby). Jej zadaniem jest udostępnić wstrzykiwaną zależność programiście w postaci zmiennej instancji. Co ciekawe, zmienna ta udostępniania jest dopiero w przypadku pierwszego użycia jej nazwy w kontekście danego obiektu. Dzieje się tak dzięki zastosowaniu metody **instance_variable_set** w bloku metody **define_method**.

Instancja ta zostanie utworzona tylko za pierwszym razem dzięki użyciu metody **instance_variable_defined?,** która zwraca **true** w przypadku gdy zmienna instancji już istnieje oraz **false** w przeciwnym wypadku. Dostęp do zmiennej instancji realizowany jest poprzez metodę **instance_variable_get**.

Jak widać dynamika języka Ruby oraz zalety metaprogramowania, pozwalają w bardzo prosty sposób dodać wymaganą zależność do obiektu. Przykład 77 przedstawia najprostszy sposób wykorzystania omawianej biblioteki dla wstrzykiwania pojedynczej zależności do tworzonego obiektu, a następnie użycie jej funkcjonalności.

Przykład 77:

```
module Validations
	class VmValidation

		NAME_PATTERN=/^[A-Za-z0-9]+$/

		def validate_name(name)
			NAME_PATTERN.match(name) ? true : false
		end
	end
end

class VmServant
	attr_injected "validations.vm_validation"

	def initialize(name)
		@name=name
	end

	def valid?
		vm_validation.validate_name(@name)
	end
end

vm_servant=VmServant.new("Debian")
vm_servant.valid?  # => true
```

Źródło: Opracowanie własne

Jest tu przedstawiony omawiany już wcześniej przypadek klasy **VmServant** oraz jej zależności **Validations::VmValidation**. Metoda **valid?** w klasie **VmServant** sprawdza poprawność nazwy zawartej w zmiennej **name**. Sprawdzanie poprawności atrybutów klasy VmServant zostało całkowicie przeniesione do klasy zależności.

Metoda **validate_name** na podstawie ustalonego wyrażenia regularnego sprawdza poprawność nazwy a następnie zwraca wartość logiczną **true**, jeżeli jest ona poprawna oraz **false**, jeżeli nie pasuje do wzorca. Widać wyraźnie nawet na tak prostym przykładzie, że zastosowanie wzorca Dependency Injection ułatwia pracę programiście.

Przykład 78 prezentuje jak mogłaby wyglądać klasa VmServant bez użycia wstrzykiwania zależności.

Przykład 78:

```
class VmServant
      def initialize(name)
            @name=name
            @vm_validation=Validations::VmValidation.new
      end
      def valid?
            @vm_validation.validate_name(@name)
      end
end
```

Źródło: Opracowanie własne

Programista musi samodzielnie utworzyć obiekt klasy **Validations::VmValidation** oraz przypisać go do zmiennej instancji. W tym przypadku nie ma potrzeby udostępniania tej zmiennej na zewnątrz obiektu. Jeżeli jednak programista chciałby to zrobić musiałby dodatkowo zadeklarować odpowiednie metody dostępowe w definicji klasy (attr_reader i attr_writer lub attr_accessor).

5.2. Przykład użycia – Serwer XML-RPC

Serwer XML-RPC jest bardzo dobrym przykładem aplikacji, w której zastosowanie wstrzykiwania zależności jest uzasadnione.

5.2.1 XML-RPC

RPC to protokół zdalnego wywoływania procedur stworzony przez firmę Sun. Jest to prosta, przenośna i niezależna od języka oprogramowania technologia komunikacji rozproszonych systemów wykorzystująca protokół HTTP jako kanał transportowy a język XML do opisu danych.

HTTP jest protokołem warstwy aplikacji przeznaczonym dla rozproszonych i zorientowanych na media systemów informacyjnych. Jest to protokół bardzo ogólny, bezstanowy, który może być wykorzystany do dystrybucji informacji hipertekstowej oraz jako serwer rozproszonych zorientowanych obiektowo systemów (poprzez rozszerzenie metody request, kodów błędów oraz nagłówków).

Komunikat XML-RPC jest realizowany jako żądanie HTTP-POST. Treścią wiadomości jest struktura XML. Procedura wykonywana jest na serwerze a wynik jest zwracany jako poprawnie sformatowany XML. Parametrem procedury mogą być liczby, napisy, struktury i tablice. Sam protokół jest synchroniczny – na każde żądanie (request) generowana jest odpowiedź (response).

Przykład 79 przedstawia wywołanie XML-RPC w postaci pliku XML.

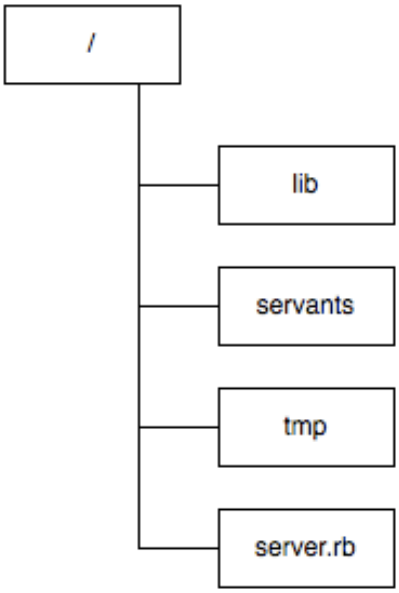
Przykład 79:

```
<?xml version=\"1.0\"?>
      <methodCall>
            <methodName>Servant.run</methodName>
            <params>
                  <param>
                        <value>
                              <i4>100</i4>
                        </value>
                  </param>
                  <param>
                         <value>Name1</value>
                  </param>
                  <param>
                        <value>Description</value>
                  </param>
            </params>
      </methodCall>
```

Źródło: Opracowanie własne

W poniższym rozdziale zostanie zaprezentowany serwer XML-RPC w języku Ruby, który sparsuje otrzymane żądanie XML i wywoła metodę w odpowiednim obiekcie. Zarządzaniem obiektami, na których możliwe będzie wywoływanie zdalnych metod zajmie się opisany wyżej kontener obiektów. Dostawcy usług (servants), czyli instancje udostępnianych obiektów nie muszą być tworzone za każdym razem, gdy nadejdzie request. Kontener Inversion of Control zapewni ich trwałość w czasie działania programu oraz dostępność ich usług. Dodatkowa funkcjonalność dostawców, taka jak logowanie czy walidacja otrzymanych danych zostanie zrealizowana poprzez wstrzykiwanie zależności.

Rysunek 16 przedstawia strukturę katalogów projektu.



Rysunek 16. Struktura katalogów w projekcie serwera XML-RPC.
Źródło: Opracowanie własne

W katalogu lib projektu znajdować się będą biblioteki niezbędne do działania serwera. Będzie to opisywana w tej pracy biblioteka realizująca wzorzec projektowy Dependency Injection dla języka Ruby, oraz prosta autorska biblioteka służąca do komunikacji protokołem XML-RPC.

W katalogu servants znajdować się będą pliki z definicjami dostawców usług (servants), którymi zarządzał będzie kontener Inversion of Control. Jest to katalog, którego zawartość opisana została wcześniej w tym rozdziale.

Katalog tmp to miejsce przechowywania tymczasowych plików takich jak numer procesu serwera XML-RPC.

Plik server.rb to główny plik projektu. Jest to punkt wejściowy do programu, dołączane są w nim wymagane biblioteki oraz inicjalizowana jest instancja kontenera wraz z jego zawartością. Jest to również pewnego rodzaju uchwyt dla serwerów HTTP, które będą obsługiwać opisywany tutaj serwer XML-RPC.

5.2.2 LibXML

Prace nad serwerem XML-RPC należy rozpocząć od parsowania przychodzących żądań XML. W opisywanym projekcie zastosowano bibliotekę LibXML, która jest portem biblioteki dla języka C i służy do obsługi dokumentów standardu XML. Zostala napisana przez Daniela Veillarda i pierwotnie przystosowana była dla środowiska GNOME. Udostępniana jest na licencji X11.

Istnieje wiele bibliotek XML dla języka Ruby. Wybór LibXML podyktowany był dużo większą wydajnością oraz szybkością obsługi dokumentów XML. Biblioteka ta dostarcza oba bardzo popularne interfejsy programistyczne do sekwencyjnego parsowania dokumentów XML. Są to SAX (Simple API for XML) oraz DOM (Document Object Model).

SAX jest pareserem operującym na strumieniach, który obsługiwany jest przy pomocy specjalnie przygotowancyh zdarzeń. Zdarzenia te są metodami programistycznymi, które przeprowadzają operacje na pojawiających się danych z dokumentu XML. SAX potrafi rozpoznać między innymi węzły tekstowe oraz instrukcje przetwarzania. Zdarzenia obejmują elementy SAX zarówno na ich początku jak i końcu. Cechą charakterystyczną SAX jest jego jednokierunkowość. Dane, które już raz zostały przetworzone, nie mogą zotać ponownie użyte. Oznacza to, że w przypadku parsowania dużych dokumentów XML ich elementy nie są przetrzymywane w pamięci operacyjnej komputera – dostaje się do niej tylko to, co jest potrzebne programiście. Opisywany w tym rozdziale serwer XML-RPC wykorzystuje interfejs programistyczny SAX do parsowania żądań RPC.

5.2.3 Rack

Rack jest modularnym interfejsem programistycznym do tworzenia aplikacji HTTP w języku Ruby. Poprzez opakowanie żądań HTTP w proste i przyjazne API, dostarcza funkcjonalność protokołu HTTP do budowania aplikacji webowych. Jest udostępniany na licencji MIT i pozwala na integrację aplikacji w Ruby z popularnymi serwerami HTTP takimi jak Apache, Lighttpd czy Nginx. Przykład 80 prezentuje plik server.rb – punkt wejściowy do programu serwera XML-RPC, który jest realizacją interfejsu Rack.

Przykład 80:

```
$: << ::File.expand_path(::File.dirname(__FILE__)+"/lib")
require 'rpc/server'
require 'sunscreen/sunscreen'

$registry=Sunscreen::Registry.new
```

```
$registry.load(::File.expand_path(::File.dirname(__FILE__) +
'/servants'))

server=RPC::Server::Server.new

run server
```

Źródło: Opracowanie własne

Zmienna **$:** jest tablicą ścieżek systemowych, która jest przeszukiwana podczas wywoływania metody require. W tym przypadku do tej tablicy dodawana jest ścieżka do katalogu projektowego lib, w którym znajdują się biblioteki serwera XML-RPC oraz kontenera Inversion of Control.

Po dodaniu do programu wymaganych bibliotek następuje utworzenie instancji kontenera podając mu ścieżkę do katalogu z plikami definiującymi dostawców usług (**servants**). Kontener jest deklarowany jako zmienna globalna tak, aby był dostępny w każdym miejscu programu.

Następnie tworzona jest instancja serwera XML-RPC, która jest rejestrowana jako uchwyt HTTP poprzez wywołanie metody **run**. Od tego momentu obiekt klasy RPC::Server::Server staje się odbiorcą żądań HTTP poprzez metodę call, która musi zostać zaimplementowana. Przykład 81 prezentuje klasę RPC::Server::Server.

Przykład 81:

```
module RPC
module Serwer

class Server
      def call(env)
            @request=Rack::Request.new(env)
            callback=parse(@request.env["rack.input"].read)

            method_call=callback.method_name.split('.')

            servant_class=get_servant(method_call)

            operation=get_operation(method_call)

            servant=$registry.lookup(servant_class)

            resp=eval("servant.#{operation}(#{callback.params.collect
            {|item| "'#{item}'"}.join(",")})")

            [200, { "Content-Type" => "text/html" }, [resp]]
      end

      def parse(xml)
            parser = LibXML::XML::SaxParser.string(xml)
            callback=Callbacks.new

            parser.callbacks = callback
            parser.parse
            return callback
      end
end
end
end
```

Źródło: Opracowanie własne

Argumentem wywołania metody call jest środowisko serwera **HTTP**, czyli wszystkie nagłówki, dane nadawcy żądania oraz zawartość żądania.

Dokument przesyłany protokołem HTTP zawarty jest w zmiennej środowiskowej **rack.input.** Jej wartość przekazywana jest do metody **parse**, która inicjalizuje interfejs programistyczny **SAX** poprzez utworzenie instancji klasy **LibXML::XML::SaxParser**. Następnie tworzony jest obiekt reprezentujący obsługę zdarzeń dla parsera **SAX**.

W wyniku działania parsera zwracane są informacje o dostawcy usług, metodzie wywoływanej na obiekcie dostawcy oraz jej parametrach. Po rozłożeniu otrzymanego ciągu znaków reprezentującego wywołanie metody, tworzone są zmienne **servant_class** – przechowująca klasę dostawcy oraz **operation** - przechowująca nazwę metody. W tym momencie zostaje pobrany obiekt dostawcy z kontenera Inversion of Control. Wykorzystana zostaje jedynie funkcjonalność pozwalająca zarządzać obiektami a nie wstrzykiwaniem zależności między nimi.

Mając już instancję obiektu dostawcy można bezpiecznie wywołać na nim wymaganą metodę. Zostanie to przeprowadzone przy pomocy metody **eval,** która wykonuje ciąg znaków tak jakby był to kod programu. Lista argumentów metody zawarta jest w zmiennej **callback.params**, której elementy są dynamicznie łączone znakiem przecinka.

Jest to minimum wymagane do działania serwera XML-RPC opartego o dostawców usług zarządzanych przez kontener Inversion of Control. Użycie tej biblioteki znacznie upraszcza budowanie takiego serwera dzięki zautomatyzowanemu dostępowi do usług oraz metod.

5.2.4 Dostawca VmServant

Przykładowy dostawca usług to obiekt klasy VmServant. Reprezentuje on dostęp do wirtualnej maszyny (stąd VM – Virtual Machine) w środowisku zarządzania wirtualnymi systemami operacyjnymi XEN.

Przykład 82 prezentuje prostą metodę deploy, służącą do utworzenia maszyny w środowisku wirtualizacji.

Przykład 82:

```
require "xen/vm"
class VmServant

      attr_injected "validations.vm_validation"
      attr_injected "system.logger"

      def deploy(template_id,name,description)
            raise "Validation Error" if
            !vm_validation.validate_deploy(template_id)

            response=XenVM.deploy(template_id,name,description)

            if response.success?
                  return true
                  logger.info("Deployment successful")
            else
                  raise "Deployment Error"
            end
      end
end
```

Źródło: Opracowanie własne

Można zaobserwować, że dostawca **VmServant** posiada zadeklarowane zależności służące do walidacji oraz logowania zdarzeń. Zależności te są automatycznie wstrzykiwane przez kontener Inversion of Control przy pierwszym użyciu obiektu. Następnie zależności te użyte są w przypadku metody **deploy** do sprawdzenia poprawności numeru szablonu wirtualnej maszyny oraz do zalogowania informacji o poprawnym założeniu nowego systemu.

Metoda **deploy** jest bardzo krótka. Programista nie musi zajmować się tworzeniem obiektów, które tak naprawdę na stałe są związane z klasą VmServant. Dzieje się to automatycznie przy użyciu kontenera Inversion of Control.

Mając utworzoną definicję klasy dostawcy (VmServant) serwer XML-RPC jest gotowy do użycia. Aplikację korzystającą z interfejsu programistycznego Rack można uruchamiać przy pomocy polecenia rackup (w środowisku testowym) lub z wykorzystaniem, któregoś z popularnych serwerów HTTP. Gdy serwer XML-RPC jest już dostępny w sieci, można się do niego łączyć przy pomocy żądań HTTP-POST. Przykład 83 prezentuje sposób połączenia się do serwera w języku Ruby.

Przykład 83:

```
require 'net/http'
nex = Net::HTTP.new("127.0.0.1",8888)
headers = {'Content-Type' => 'text/xml'}
resp,data = nex.post("/", testxml, headers)
```

Źródło: Opracowanie własne

Konstruktor obiektu klasy Net::HTTP przyjmuje dwa parametry – **host** oraz **port**. Aby poprawnie wysłać żądanie XML-RPC należy zadeklarować typ standardu wymiany danych **MIME** (Multipurpose Internet Mail Extensions). Jest on deklarowany w nagłówkach żądania HTTP.

Dla XML-RPC wymaganym content-type MIME jest **text/xml.** Po utworzeniu tablicy asocjacyjnej reprezentującej nagłówki żądania należy je wysłać przy pomocy metody **post** instancji klasy Net::HTTP. Przyjmuje ona 3 parametry. Pierwszym jest url do zasobu zdalnego (w przypadku serwera XML-RPC jest to katalog główny /). Drugi to treść przesyłanych danych (dokument XML), a trzeci to nagłówki.

Przykład 84 prezentuje dokument w formacie XML zgodny z protokołem XML-RPC służący do wywołania metody **deploy** na dostawcy klasy **VmServant**.

Przykład 84:

```
<?xml version=\"1.0\"?>
<methodCall>
      <methodName>VmServant.deploy</methodName>
      <params>
            <param>
                  <value>
                        <i4>908</i4>
                  </value>
            </param>
            <param>
                  <value>Debian 1</value>
            </param>
            <param>
                  <value>Database Replication Server</value>
            </param>
      </params>
```

```
</methodCall>
```

Źródło: Opracowanie własne

Metoda deploy wywoływana jest z trzema parametrami. Pierwszym jest wartość numeryczna przedstawiająca numer szablonu systemu operacyjnego (przygotowana przez dostawcę usług wirtualizacyjnych wersja systemu gotowa do pracy). Drugi argument to podana przez użytkownika nazwa nowego systemu, a trzeci to jego krótki opis (przeznaczenie).

Tak przygotowany dokument XML spowoduje założenie wirtualnej maszyny w systemie XEN przez opisanego powyżej dostawcę VmServant zarządzanego przez kontener Inversion of Control a wywołanego przez serwer XML-RPC.

5.3. Podsumowanie

Implementacja wzorca Dependency Injection w języku Ruby nie wymaga dużego nakładu pracy. Dynamika języka oraz zalety metaprogramowania pozwalają na dynamiczne dodawanie zależności do istniejących już komponentów. Powyższy rozdział szczegółowo prezentuje sposób implementacji kontenera Inversion of Control oraz idei wstrzykiwania zależności. Przykład użycia przygotowanej biblioteki to minimalistyczny serwer XML-RPC realizujący dostęp do systemu wirtualizacji XEN.

Użycie wstrzykiwania zależności oraz kontroli nad obiektami przez kontener pozwala programiście skoncentrować się na tym, co ważne – na implementacji samego protokołu. Powoduje to, że pracuje on szybciej i ma dużo mniej kodu do utrzymania w przyszłości.

Prezentowana w tym rozdziale biblioteka czerpie z to, co najlepsze w opisanych w tej pracy istniejących już rozwiązaniach w języku Java. Największy wpływ na ostateczny sposób działania kontenera miały doświadczenia autora w pracy z bibliotekami Google Guice oraz Spring Framework.

Mimo ich uniwersalności, prezentowane rozwiązanie koncentruje się jedynie na jednym rodzaju wstrzykiwania zależności. Jest to Setter Dependency Injection. Powodem tego jest specyfika języka Ruby. Brak zmiennych finalnych (final) powoduje, że wstrzykiwanie zależności poprzez konstruktor nie różniłoby się czasem wykonania ani nakładem pracy od ręcznego tworzenia obiektów przez programistę. Konstruktor Dependency Injection nie miałby również sensu z powodu braku możliwości skonfigurowania zależności pomiędzy komponentami w równie prosty sposób jak zostało to zrobione w przypadku metod dostępowych, gdzie informacja o tym, co ma być wstrzyknięte jest konfigurowana w samej nazwie zależności.

6. Wady, zalety oraz perspektywy rozwoju

Prezentowana w tej pracy biblioteka realizująca wzorzec Dependency Injection posiada kilka wad wynikających z przyjętego podejścia do wstrzykiwania zależności. Mimo iż dynamika języka Ruby pozwala na dosyć dużą dowolność przy budowaniu tego typu narzędzi, nie udało się uniknąć pewnych różnic funkcjonalnych w stosunku do prezentowanych w rozdziale II implementacji w języku Java. Prostota końcowego rozwiązania wiąże się z pewnymi niedogodnościami, które mogą wydać się istotne przy próbie zaawansowanego użycia biblioteki w środowisku produkcyjnym.

Jedną z niedogodności może się wydać brak Constructor Dependency Injection. Wstrzykiwanie zależności poprzez konstruktor wydało się zbędne w przypadku języka Ruby. Główną zaletą tego typu Dependency Injection w implementacjach w języku Java była możliwość deklarowania zmiennych finalnych jako zależności. W Ruby one nie występują. Takie swoiste zamrażanie stanu obiektu i jego zależności kłóciłoby się z założeniami dynamiki języka programowania i byłoby tak naprawdę tworem sztucznym. W związku z tym proponowane rozwiązanie całkowicie odchodzi od tego typu wstrzykiwania zależności koncentrując się całkowicie na operacjach dostępowych, których przeróżne warianty są możliwe przy użyciu metaprogramowania.

Kolejną niedogodnością w pewnych przypadkach jest sposób zarządzania obiektami przez kontener Inversion of Control. Przeszukuje on podany katalog rekursywnie w poszukiwaniu definicji klas i modułów. Niepożądanym działaniem byłoby gdyby w tym katalogu znajdowały się definicje klas, które nie powinny się znaleźć w kontenerze.

Decyzja o utworzeniu oddzielnego miejsca w projekcie na klasy kontenera zależy od programisty i nie jest w żaden sposób wymuszana przez bibliotekę. Od zdrowego rozsądku twórcy projektu zależy jego struktura. Potencjalnym rozwiązaniem problemu tworzenia obiektów niechcianych klas przez kontener jest oznaczanie ich w pewien sposób. Może się to odbywać poprzez dodawanie pewnego modułu do takiej klasy. Odpowiada to implementowaniu przez klasę interfejsu w Javie. Kontener przeszukując folder projektu, tworzyłby instancje jedynie tych klas, które posiadają dołączony właściwy moduł.

Zdecydowaną zaletą prezentowanej w tej pracy biblioteki Dependency Injection jest brak konieczności definiowania połączeń pomiędzy komponentami jak to miało miejsce w implementacjach w języku Java. Nazwa zależności jest sama w sobie informacją dla kontenera mówiącą o tym, jaka klasa za nią odpowiada i do jakiej zmiennej powinna zostać wstrzyknięta. Przyjęte nazewnictwo pozwala wstrzykiwać zależności umieszczone w dowolnie zagnieżdżonych zakresach nazw.

Wiąże się z tym jednak pewien problem. Gdy programista chce dodać dwie zmienne oznaczające zależność tej samej klasy biblioteka utworzy tylko jedną zmienną. Dzieje się tak, ponieważ nie ma możliwości stworzenia aliasu dla nowej zmiennej, której instancja już istnieje i jest dołączona do obiektu. Metody attr_injected oraz attr_injected_new mogłyby zostać w przyszłości rozszerzone o możliwość definiowania dowolnej nazwy zmiennej dla wstrzykiwanej zależności. Mogłoby się to odbywać poprzez zmianę atrybutów tych metod na tablice asocjacyjne, których klucze są dotychczasowymi odpowiednio przygotowanymi nazwami klas, natomiast wartości to dowolne nazwy dla zmiennych ustalone przez programistę.

Jak widać język Ruby oraz struktura biblioteki (nadpisana klasa Object) pozwalają na bardzo łatwe jej rozszerzenie o nową funkcjonalność. Integracja biblioteki z zaprezentowanym przykładem użycia – serwerem XML-RPC, pozwoliłaby na udostępnienie obiektów pomiędzy kontenerami.

Wymagałoby to pewnej ilości konfiguracji między systemami i pozwoliłoby wstrzykiwać zależności, które są instancjami klas nieistniejącymi w lokalnym kontenerze obiektów. Kontener połączony ze zdalnymi systemami tego samego typu (implementującymi opisywaną w tej pracy bibliotekę) byłby w stanie wyszukiwać oraz wstrzykiwać ich obiekty tak samo jak lokalne. Obsługa zależności nadal polegałaby na użyciu metod attr_injected oraz attr_injected_new. Kontener sam decydowałby czy obiekt znajduje się w systemie lokalnym czy zdalnym. Oczywiście takie zdalne obiekty nie miałyby pełnej tożsamości obiektu. Byłaby to jedynie pewna abstrakcyjna klasa realizująca żądania do obiektu poprzez wywołanie metody method_missing.

Język Ruby pozwala na wykonywanie na obiektach metod, które tak naprawdę nie istnieją. W takim przypadku obiekt zawsze wywołuje metodę method_missing.

Jest to kolejna zaleta metaprogramowania. Ruby pozwala programiście decydować o obsłudze zdarzeń, które w każdym innym wypadku spowodowałyby wyjątek. Przy użyciu tej metody, wystarczyłoby utworzyć pewną pustą klasę, która przechwyci wywołania metod i wyśle je do serwera XML-RPC. W ten sposób otrzymamy niewidoczną warstwę przekazywania żądań do obiektów. Takie rozproszenie funkcjonalności pomiędzy zdalne systemy jest dodatkowym aspektem odwrócenia sterowania. System w takiej postaci działałby podobnie jak znane z Javy EJB.

Kolejnym możliwym rozszerzeniem biblioteki jest dostarczenie razem z nią przygotowanych klas, które realizują pewną specyficzną funkcjonalność i mogą być wstrzykiwane do obiektów użytkownika dokładnie w ten sam sposób jak jego własne zależności. Pozwala to stworzyć pewien specyficzny system wtyczek (plugins), które mogłyby być dostarczane w ramach biblioteki lub jako osobne dodatki. Taką wtyczką mogłaby być klasa logująca zdarzenia do pliku lub bazy danych. Jeżeli posiadamy funkcjonalność bazy danych to biblioteka może dostarczać klasy pełniące funkcję uchwytów do różnych baz danych. Pozwoli to tworzyć tylko jeden obiekt realizujący połączenie z bazą.

Podstawowa funkcjonalność prezentowanej biblioteki jest na tyle uniwersalna, że wtyczki mogłyby posiadać wstrzykiwane zależności na tej samej zasadzie jak obiekty użytkownika.

Pewnej dyskusji oraz zastanowienia wymagałoby zadeklarowanie oddzielnego katalogu dla tych predefiniowanych komponentów (prawdopodobnie plugins). W takim jednak wypadku kontener musiałby operować na dwóch oddzielnych katalogach i wymieniać między nimi informacje o zależnościach, co burzy ideę jednego katalogu obiektów. Być może lepszym rozwiązaniem byłoby utworzenie pewnego zakresu nazw w katalogu użytkownika, co pozwoli na łatwe odróżnienie komponentów systemowych od komponentów utworzonych przez programistę.

Opisywana w tej pracy implementacja wzorca Dependency Injection w języku Ruby stanowi tak naprawdę bazę dla bardziej rozbudowanego systemu zarządzania obiektami.

W przypadku wprowadzenia biblioteki w środowisku programistów języka Ruby, możliwość rozbudowy oraz działania na zasadzie wtyczek pozwoliłyby na stworzenie pewnej społeczności wokół prezentowanego projektu.

Od początku tworzenia biblioteki jest ona poddana zarządzaniu wersjami (system Subversion). Istnieje możliwość otworzenia repozytorium i ustanowienia biblioteki jako Open Source. Wówczas każdy chętny programista mógłby dowolnie zmieniać i wykorzystywać kod programu do swoich celów. Spektrum projektów, w których można wykorzystać prezentowaną bibliotekę jest bardzo szerokie (właściwie nieograniczone). W obecnej formie implementacja Dependency Injection mogłaby z powodzeniem zostać użyta zarówno w prostych aplikacjach lub skryptach jak i w rozbudowanych systemach rozproszonych (z wykorzystaniem zaprezentowanego serwera XML-RPC).

Decyzja o przydatności biblioteki w konkretnym projekcie zależy oczywiście od uznania programisty. Można się spotkać z opinią, że kopiowanie rozwiązań z języka Java do Ruby nie ma uzasadnienia i jest bezcelowe. Twierdzi się, że Ruby jest na tyle dynamicznym językiem, że nie potrzebuje tak rozbudowanych wzorców projektowych jak Dependency Injection. Myślę, że powyższa praca pokazuje, iż wykorzystanie wspomnianej dynamiki języka Ruby oraz technologii metaprogramowania może być użyte zgodnie z charakterem języka i może być pozbawione nawyków z innych języków programowania. W rozbudowanych systemach zaprezentowany sposób zarządzania zależnościami niewątpliwie przyczyni się do ułatwienia pracy programistom i do mniejszej ilości kodu wymaganego do stworzenia i konserwacji.

Prace cytowane

[1]. R. Vanbrabant. Google Guice: Agile Lightweight Dependency Injection Framework. Apress, 2008.

[2]. R. Harrop, J. Machacek. Pro Spring. Apress, 2005

[3]. D. R. Prasanna. Dependency Injection. Manning, 2009

[4]. D. Thomas. Programming Ruby: The Pragmatic Programmers' Guide. The Pragmatic Programmers', 2006

[5]. M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html, Jan 2004.

[6]. A. Pohl. Typy danych. http://apohllo.pl/dydaktyka/ruby/intro/typy-danych, 2007

[7]. A. Pohl. Metaprogramowanie. http://www.apohllo.pl/dydaktyka/ruby/intro/metaprogramowanie, 2008

[8]. What is Avalon-Framework http://wiki.apache.org/excalibur/AvalonFramework

[9]. P. Bellavista. Lightweight Container: tecnologia Spring http://lia.deis.unibo.it/Courses/sd0809-info/lucidi/07-Spring%281x%29.pdf

Spis rysunków:

Spis tabel: