ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 2012
Seria: ORGANIZACJA I ZARZĄDZANIE z. 61 Nr kol. 1873

Ireneusz J. JÓŹWIAK, Przemysław SASNAL
Politechnika Wrocławska
Wydział Informatyki i Zarządzania

STRATEGIA ODWRÓCENIA STEROWANIA ZALEŻNOŚCIAMI W JĘZYKU PROGRAMOWANIA GO

Streszczenie. W artykule przedstawiono analizę sposobów odwrócenia sterowania zależnościami w języku programowania Go. Rozpatrzono dostępne sposoby implementacji rozwiązania oraz konfiguracji używanego kontenera odwrócenia sterowania. Opisano korzyści wynikające z odwrócenia sterowania zależnościami w projekcie informatycznym.

INVERSION OF CONTROL OF DEPENDENCIES STRATEGY IN GO PROGRAMMING LANGUAGE

Summary. The paper presents ways of inversion of dependencies control in Go programming language. Available ways of implementing solution and configuration of IoC container has been considered. Authors describe also advantages of using inversion of dependencies of control in IT project.

1. Wprowadzenie

Odwrócenie sterowania zależnościami w języku programowania jest strategią, polegającą na przeniesieniu odpowiedzialności dotyczącej tworzenia i łączenia ze sobą komponentów na zewnątrz obiektów. Podejście takie ma niewątpliwe zalety, takie jak zmniejszenie kosztów utrzymania oprogramowania przez lepszą testowalność kodu oraz łatwiejsze ponowne użycie komponentów dzięki ich wysokiej spójności. Odwrócenie sterowania zależnościami ułatwia także wdrożenie praktyk znanych z metodyk wytwarzania oprogramowania, jak np. Test-Driven Development.

Z tego też powodu hasła takie jak „odwrócenie sterowania" (ang. *Inversion of Control*) czy „wstrzykiwanie zależności" (ang. *Dependency Injection*) są w ostatnich latach bardzo popularne w społecznościach programistów języków obiektowo zorientowanych, głównie Javy i C#. Ponadto idee te zyskują duże zainteresowanie wśród programistów korzystających z języków niebędących typowymi językami obiektowo zorientowanymi, ze statycznym typowaniem i sztywnymi regułami dotyczącymi hierarchii klas, deklaracji typów i interfejsów. Frameworki umożliwiające odwrócenie sterowania aspektem zarządzania zależności istnieją także w językach łączących w sobie inne paradygmaty programowania poza obiektowym, często posiadających dynamiczne typowanie, jak np. Ruby czy Python. Jednak w trakcie badań autorzy artykułu nie znaleźli informacji o jakimkolwiek projekcie, który adresowałby zagadnienie odwrócenia sterowania zależnościami w języku programowania Go.

Niniejszy artykuł opisuje prace nad możliwością odwrócenia sterowania zależnościami w języku programowania Go.

2. Korzyści wynikające z odwrócenia sterowania zależnościami

W rozwiązaniach dla biznesu kluczowe stają się czynniki, które sprawiają, że oprogramowanie jest opłacalne. Jeśli system komputerowy jest skomplikowany i ma funkcjonować przez długi okres czasu, koszty utrzymania tego systemu stają się kluczowe. W dobrze zaprojektowanym systemie kontrakt pomiędzy klientem, czyli komponentem korzystającym z pewnych funkcjonalności, a serwisem, czyli komponentem je dostarczającym, scharakteryzowany jest za pomocą zbioru metod, jakie można wykonać na danym obiekcie. Odizolowanie konkretnej implementacji od jej deklaracji sprawia, że architekt, projektując system, nie musi analizować szczegółów implementacji serwisu, lecz jedynie zbiór usług, jakie on dostarcza. Umożliwia to również zastosowanie wielu specyficznych implementacji pewnego generycznego serwisu, które mogą być użyte przy budowie różnych serwisów. Dzięki zdefiniowaniu sztywnego kontraktu pomiędzy klientem a serwisem znacząca zmiana w wewnętrznej implementacji serwisu ma minimalny wpływ na klienta, tak że obaj mogą się rozwijać niezależnie, dopóki kontrakt jest spełniony [6]. Zaprojektowanie systemu w sposób modułowy powoduje, że komponenty mogą być ponownie używane oraz łatwy jest podział kodu na uporządkowane części, które można łatwiej utrzymywać ze względu na ich wysoką spójność.

Jednym z problemów, jaki napotykają programiści, tworząc serwisy, jest konieczność połączenia w jedną całość wielu różnych serwisów tworzących system lub komponent, by wiedziały o sobie jak najmniej (w celu ograniczenia niepotrzebnych zależności). Przy

stosowaniu „tradycyjnego" podejścia, w którym zadaniem klienta jest utworzenie wszystkich serwisów, z których korzysta (co zaleca wzorzec Creator, jeden z wzorców GRASP [4]), system jest nieelastyczny, tzn. jakakolwiek próba podmiany serwisu na inną jego implementację skutkuje koniecznością zmian w kodzie klienta (ponieważ to właśnie tam są instancjonowane i konfigurowane serwisy). Jak widać, konieczne jest więc rozdzielenie kodu logiki biznesowej (czyli kodu odpowiedzialnego za realizację konkretnych zadań i operacji) od kodu infrastruktury (czyli od wszystkiego, co służy wsparciu funkcji biznesowych, instancjonowaniu serwisów i składaniu ich w całość, uzyskiwaniu połączeń do baz danych itp.). Odwrócenie sterowania zależnościami właśnie rozwiązuje ten problem, ponieważ część klas odpowiedzialna za instancjonowanie obiektów jest przeniesiona do biblioteki często nazywanej kontenerem IoC. Zdaniem Dhanjiego Prasanny oddzielenie tych dwóch warstw „nie tylko pomaga utrzymać logikę aplikacji przejrzystą i skupioną na celu, lecz również zapobiega testom byciu zanieczyszczonymi przez odwracające uwagę błędy, które mogą nie mieć nic wspólnego z celem kodu" [6].

Jednym z następstw modułowej budowy jest możliwość łatwego testowania oprogramowania. Obecnie w procesie rozwijania oprogramowania bardzo duży nacisk jest kładziony na jego utrzymywanie, ponieważ to ono generuje największe koszty [1]. Dlatego też należy zwrócić uwagę na dobre zdefiniowanie procesu sprawdzania jakości oprogramowania oraz łatwość jego stosowania. Dzięki modułowej budowie i oddzieleniu kodu infrastruktury od logiki biznesowej tak samo łatwo, jak można podmieniać różne implementacje serwisów, można wykorzystać w aplikacji serwisy testowe udające prawdziwe serwisy, lecz znacznie od nich prostsze (często wręcz trywialne). W anglojęzycznej literaturze serwisy testowe są nazywane mockami (ang. „mock" znaczy „atrapa"). Atrapy serwisów pozwalają symulować zachowanie złożonych serwisów z wieloma wzajemnymi zależnościami, odtwarzając operacje zgodne z oczekiwaniami programisty, wynikającymi ze scenariusza testowego. Pozwala to na ich jednostkowe testowanie w izolacji, co jest jednym z kluczowych aspektów metodyk wytwarzania oprogramowania, w szczególności implementacji kierowanej testami (ang. *Test-Driven Development – TDD*) [5].

Odwrócenie sterowania zależnościami powoduje również zmniejszenie ilości kodu infrastruktury, który trzeba utrzymywać. Jest to powodowane faktem, że programiści zazwyczaj nie muszą pisać kodu tworzącego instancje oraz wiążącego je ze sobą, a nawet, jeśli jest to konieczne, w niektórych implementacjach muszą oni wówczas utworzyć bierny kod źródłowy, niewykonujący żadnej logiki, który jest dużo łatwiejszy do utrzymania [7].

3. Odwrócenie sterowania zależnościami w języku programowania Go

Sprecyzowanie wymagań dotyczących biblioteki realizującej odwrócenie sterowania zależnościami dla języka Go jest dość trudną kwestią. Należy rozważyć wiele sposobów implementacji rozwiązania. Niektóre z opcji mogą okazać się możliwe do zrealizowania, jednak ich charakter będzie nieodpowiedni do specyfiki języka Go. Kwestie te zostaną przeanalizowanie w tym rozdziale.

Do odwrócenia sterowania zależnościami można wykorzystać kilka wzorców projektowych: fabrykę abstrakcyjną, Service Locator czy wstrzykiwanie zależności [6]. Po ich przenalizowaniu, autorzy artykułu uznali wzorzec wstrzykiwania zależności (ang. *Dependency Injection*) za najlepszy do tego celu. Istotą tego wzorca jest odwrócenie ról i przekazywanie komponentom referencji do zależnych serwisów za pomocą metod dostępowych, konstruktorów, metod interfejsów i innych sposobów w zależności od użytego idiomu. Zależności te przekazywane są z zewnątrz przez tzw. kontener IoC.

Wzorzec ten nie wymaga od programisty ręcznego łączenia ze sobą serwisów oraz tworzenia dużej ilości niepotrzebnego kodu, jak to ma miejsce w przypadku stosowania wzorca fabryki abstrakcyjnej, który „dostarcza interfejsu do tworzenia rodzin związanych ze sobą lub zależnych obiektów bez specyfikowania ich konkretnych klas" [6], co jest realizowane przez implementację statycznych metod. Wstrzykiwanie zależności jest dużo bardziej elastycznym rozwiązaniem niż implementacja osobnych fabryk dla różnych możliwych wariantów serwisów.

Innym rozważanym wzorcem był Service Locator. Jego istota polega na zaprojektowaniu specjalnego rejestru, którego zadaniem jest odnalezienie w czasie inicjalizacji komponentu jego zależnych serwisów, opublikowanych wcześniej przez zewnętrzne źródła. Zachowanie to jest propagowane rekurencyjnie w dół grafu zależności przez wszystkie komponenty aż do zbudowania poprawnego serwisu. Jednak dużym minusem użycia wzorca Service Locator w stosunku do wzorca wstrzykiwania zależności jest konieczność modyfikacji kodu źródłowego komponentów w przypadku zamiany lub refaktoryzacji biblioteki realizującej zadania kontenera IoC. Dzięki użyciu wstrzykiwania zależności komponenty są bardziej uniwersalne i łatwiej mogą być ponownie wykorzystane w innych projektach, gdyż nie występują w obiektach odwołania do serwisu będącego rejestrem komponentów.

Po wybraniu wzorca wstrzykiwania zależności do zaimplementowania w projekcie naturalną koleją rzeczy jest wybór zastosowanego idiomu, ponieważ wzorzec ten może być realizowany na wiele sposobów.

Wstrzykiwanie zależności przez interfejsy (ang. *Interface Dependency Injection*) polega na używaniu do przekazywania referencji metod dostępowych, zdefiniowanych w osobnych interfejsach, umożliwiających przekazanie referencji dla każdego typu zależności. Jednak ten idiom niepotrzebnie wprowadza dużą rozwlekłość w kodzie, ponieważ każda możliwość wstrzyknięcia musi być w tym celu oznakowana specjalnie stworzonym interfejsem. Jest to przeciwieństwo natury języka Go, który – nie posiadając nawet hierarchii klas – jest doskonałym przykładem minimalistycznego podejścia jego twórców.

Z kolei kontekstowe wyszukiwanie zależności (ang. *Contextualized Dependency Lookup*), jako połączenie wzorca projektowego Service Locator oraz idiomu wstrzykiwania przez interfejsy, które zostały uznane za nieodpowiednie dla tego rozwiązania, również nie jest dobrym rozwiązaniem z powodu wad ich obu.

Rozpatrywano także idiom wstrzykiwania zależności przez konstruktory (ang. *Constructor Dependency Injection*). Jego zalety wynikają głównie z ograniczeń konstruktorów w typowych, obiektowo zorientowanych językach programowania. Programista może w nich zakładać, iż wszystkie utworzone obiekty są w stanie uznawanym za „poprawny", ponieważ nie ma możliwości utworzenia instancji bez podania wszystkich wymaganych przez konstruktor referencji. Jednak należy tutaj zwrócić uwagę na to, że Go z założenia nie jest typowym językiem obiektowo zorientowanym, ma jedynie pewne konstrukcje pozwalające programować obiektowo. Konstruktory w nim pełnią jedynie pewną umowną rolę, w rzeczywistości są one funkcjami zdefiniowanymi dla pakietu, zwracającymi pewien obiekt; nie mają one żadnych ograniczeń, jeśli chodzi o restrykcje języka programowania. Poza tym, na chwilę obecną, interfejs programistyczny pakietu reflect, odpowiadającego za mechanizm refleksji w języku Go, jest dość okrojony i nie umożliwia wywoływania funkcji zdefiniowanych dla pakietu, pełniących rolę konstruktorów [2].

Idiomem, którego implementacja jest możliwa i niezależna od bibliotek systemowych i mechanizmów języka, jest Code Enhancement. W pewien sposób omija on te problemy, gdyż jego istota polega na automatycznym wygenerowaniu nowego kodu źródłowego i wzbogaceniu już istniejącego przed kompilacją projektu. Takie rozwiązanie jest stosowane, aczkolwiek rzadko i głównie w przypadkach środowisk, w których nie jest możliwe stosowanie mechanizmów metaprogramowania, jak np. refleksje. Powodem tego jest fakt, iż generowanie i wzbogacanie kodu źródłowego wprowadza wiele niewygód; przede wszystkim konieczne jest wprowadzenie dodatkowego kroku w procesie budowania oprogramowania, co wymusza konieczność dodatkowej konfiguracji środowiska programisty. Poza tym, w przypadku korzystania ze środowiska IDE przy tworzeniu oprogramowania, dla takiego rozwiązania wymagane jest w nim wsparcie. Z przedstawionych powodów rozwiązanie to uznano za mało interesujące do zastosowania w języku Go.

Najodpowiedniejszymi idiomami do zastosowania w projekcie wydają się być: wstrzykiwanie zależności przez metody dostępowe (ang. *Setter Dependency Injection*) oraz przez pola (ang. *Field Dependency Injection*). Pierwszy z nich polega na przekazywaniu referencji do zależności przez metody dostępowe (tzw. settery), natomiast drugi z idiomów polega na używaniu w tym celu bezpośrednio pól klasy. Zastosowanie obu z nich w języku Go jest możliwe, gdyż mechanizm refleksji pozwala na wywołania metod oraz dostęp do pól instancji w czasie działania programu [2], co jest niezbędne do zaimplementowania tych idiomów. Oba rozwiązania są do siebie podobne, ale używanie metod dostępowych do wstrzykiwania zależności pozwala na zachowanie lepszej enkapsulacji w projekcie, gdyż pozwala ukryć wewnętrzną implementację obiektów. Jednak wykorzystywanie w tym celu pól klasy jest bardzo wygodne i przejrzyste dla programisty, gdyż nie wymaga intencjonalnego tworzenia konstruktorów ani metod. Field Dependency Injection nadaje się świetnie do prostych projektów ze względu na dużą przejrzystość rozwiązania, co dobrze wpasowuje się w minimalistyczny charakter języka, którego składnia oferuje pokaźną ilość „lukru syntaktycznego", pozwalającego oczekiwaną funkcjonalność zrealizować na wiele sposobów, a jednocześnie nie tworzyć kodu, który jest zbędny. Użycie jednego z tych dwóch idiomów wydaje się zależne od specyficznych wymagań projektu: w przypadku mniej rozbudowanych systemów wstrzykiwanie zależności przez pola zapewni szybsze tworzenie i większą przejrzystość kodu źródłowego, natomiast w bardziej złożonych, obiektowych systemach wprowadzenie metod dostępowych pozwoli na lepszą enkapsulację.

Konfiguracja kontenera IoC polega na zdefiniowaniu grafu zależności serwisów, by kontener był w stanie zinstancjonować je na żądanie użytkownika. Istnieje kilka sposobów umożliwiających konfigurację kontenera IoC: za pomocą interfejsu programistycznego, przez plik konfiguracyjny, za pomocą języka domenowego, przez statyczną analizę klas lub statyczną analizę annotacji [6]. Spośród tych rozwiązań kontener IoC powinien wspierać ich podzbiór składający się z przynajmniej jednego sposobu konfiguracji.

Po dokonanej analizie, na samym początku należy odrzucić konfigurację kontenera przez statyczną analizę struktury klas, gdyż w języku Go nie występuje hierarchia typów. Podobnie należy odrzucić konfigurację przez statyczną analizę annotacji, gdyż Go w chwili pisania tego artykułu nie obsługiwał tego mechanizmu.

Sposoby konfiguracji kontenera IoC możliwe do zaimplementowania w języku Go to: interfejs programistyczny, pliki konfiguracyjne oraz język domenowy. Interfejs programistyczny jest podstawowym interfejsem konfiguracji, który powinien być rozważany, ponieważ kod źródłowy ma wsparcie kompilatora oraz ewentualnego IDE. Poza podstawową możliwością konfiguracji dodatkowo sprawdza się on również dla skomplikowanych wiązań,

które wymagają warunkowych kroków. Martin Fowler twierdzi, że interfejs programistyczny powinien być podstawowym sposobem konfiguracji kontenera IoC niezależnie od ewentualnych, innych dostępnych sposobów jego konfiguracji [3].

Możliwość konfiguracji przez zewnętrzny plik, np. XML, jest przydatnym narzędziem w dość rozbudowanych projektach z rozbudowaną hierarchią klas. Według Fowlera, dla prostego projektu, w którym nie ma wielu wariantów wdrożenia, kilka linii kodu może być bardziej czytelne niż osobny plik XML. Poza tym konfiguracja za pomocą pliku nie daje tak szerokich możliwości jak kod źródłowy [3]. Można wysunąć z tego wniosek, że zastosowanie XML jako dodatkowego sposobu konfiguracji kontenera IoC powinno być zależne od specyfiki projektu. Warto jednak dodać, że nie jest to rozwiązanie niezbędne; interfejs programistyczny jest często jedynym sposobem konfiguracji kontenera IoC, stosowanym przez wiele projektów open-source (np. popularny projekt PicoContainer, realizujący odwrócenie sterowania zależnościami dla języka Java).

Użycie języka domenowego, zbudowanego na bazie języka programowania ma natomiast sens tylko wówczas, gdy taki zabieg uprości przejrzystość rozwiązania. Projektowane rozwiązanie ma w zamierzeniu mieć minimalistyczny charakter, pasujący do natury języka Go, więc należy wątpić, czy rzeczywiście wprowadzenie dodatkowego języka domenowego w jakikolwiek sposób uprościłoby odwrócenie sterowania zależnościami w Go. Wręcz przeciwnie – może je skomplikować.

4. Podsumowanie

W artykule przeanalizowano różne możliwości zastosowania reguły odwrócenia sterowania zależnościami w języku Go. W wyniku przeanalizowania specyfiki tego języka programowania oraz dostępnych możliwości zdecydowano się na zaproponowanie wzorca wstrzykiwania zależności jako sposobu na zaimplementowanie tego rozwiązania. Za najbardziej odpowiedni sposób przekazywania referencji przez kontener IoC uznano metody dostępowe i pola klas, do konfiguracji zaleca się natomiast używanie interfejsu programistycznego z możliwością rozszerzenia go przez dodanie obsługi dla zewnętrznych plików konfiguracyjnych XML. Na Politechnice Wrocławskiej z sukcesem został zrealizowany prototyp biblioteki dla języka Go dla rozwiązania zasugerowanego w artykule. Zrealizowane praktyczne zastosowanie potwierdza słuszność przedstawionej w artykule analizy teoretycznej oraz świadczy o możliwości implementacji takiego rozwiązania w praktyce.

Bibliografia

1. Agarwal R.: Software Development vs. Software Maintenance. 2010, http://www.irahul.com/workzone/pm/Software_Dev_vs_Maint.pdf [dostęp: 04.02.2011].
2. Autor nieznany: Package reflect, http://golang.org/pkg/reflect/ [dostęp: 29.03.2011].
3. Fowler M.: Inversion of Control Containers and the Dependency Injection pattern. 2004, http://martinfowler.com/articles/injection.html [dostęp: 08.01.2011].
4. Larman C.: Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall, Upper Saddle River 2001.
5. Martin R.C., Martin M.: Agile Principles, Patterns, and Practices in C#. Prentice Hall, Upper Saddle River 2006.
6. Prasanna D.R.: Dependency Injection. Design Patterns using Spring and Guice. Manning, Greenwich 2009.
7. Zabkiewicz S.: Kontener Inversion Of Control dla obiektów Ruby realizujący wzorzec projektowy Dependency Injection. 2009, http://www.users.pjwstk.edu.pl/~mtrzaska/Files/PraceMagisterskie/090922-Zabkiewicz.pdf [dostęp: 18.08.2010].

Abstract

Inversion of Control principle and Dependency Injection pattern are very popular in communities of enthusiasts of object-oriented programming languages during last years. Although, these ideas are most often used in Java and other programming languages with static typing and strict rules regarding class hierarchy and interface declaration, like C# and C++, they are gaining popularity among programmers using languages, which are not typical object-oriented. Frameworks, which allow inversion of control in aspect of dependencies management, are also used in programming languages, which are multi-paradigm, often dynamically typed, like Ruby or Python.

The paper gives some basic background related to Inversion of Control principle and describes advantages of this approach. Authors also present ways of realising inversion of dependencies control in Go programming language. He is considering different design

patterns like Dependency Injection and its idioms, Service Locator or Abstract Factory. There are also different ways of configuring IoC container (programmatic API, XML files, domain language, static annotations and class-hierarchy analysis), which are analysed in the paper.