O'REILLY®

Wydanie II

Podstawy architektury oprogramowania dla inżynierów

Mark Richards
Neal Ford

Helion

Tytuł oryginału: Fundamentals of Software Architecture: A Modern Engineering Approach, 2nd Edition

Tłumaczenie: Piotr Pilch, z wykorzystaniem fragmentów poprzedniego wydania w przekładzie Leszka Sagalary

ISBN: 978-83-289-2996-8

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/poaro2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

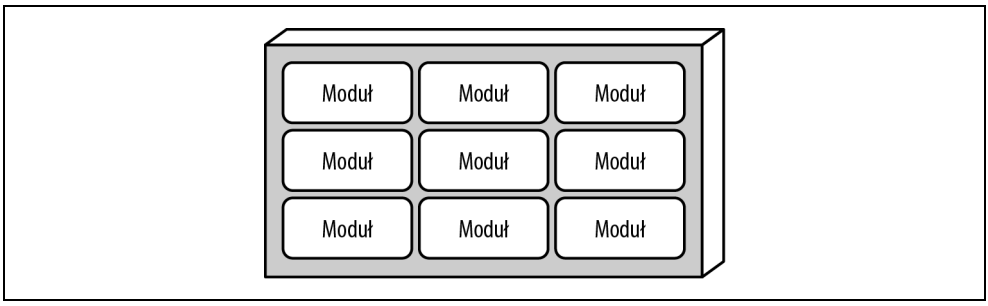
ROZDZIAŁ 11.

Styl architektoniczny monolitu modułowego

Dzięki powszechnemu zaadaptowaniu *projektowania opartego na domenie* (DDD — *Domain-Driven Design*; *https://pl.wikipedia.org/wiki/Domain-driven_design*), a także zwiększonemu naciskowi na podział domen styl architektoniczny monolitu modułowego znacząco zyskał na popularności od czasu, gdy ukazało się pierwsze wydanie tej książki (2020 r.). Z tego powodu zdecydowaliśmy się dodać do drugiego wydania rozdział opisujący (i oceniający) ten styl.

Topologia

Jak sugeruje nazwa, styl architektoniczny monolitu modułowego jest architekturą *monolityczną*. Jako taka, wdrażana jest ona jako pojedyncza jednostka oprogramowania w postaci pliku WAR (Web Archive), jednego podzespołu środowiska .NET, pliku EAR (Enterprise Archive) platformy języka Java itd. Ponieważ monolit modułowy uważa się za architekturę *dzieloną według domen* (zorganizowaną z użyciem domen biznesowych, a nie możliwości technicznych), jego izomorficzny kształt jest definiowany jako *pojedyncza jednostka wdrożeniowa z funkcjonalnością grupowaną według obszaru domenowego*. Typowa topologia monolitu modułowego jest pokazana na rysunku 11.1.



Rysunek 11.1. W przypadku stylu architektonicznego monolitu modułowego funkcjonalność jest grupowana według obszaru domenowego

Aby zrozumieć naturę ukierunkowania na domeny w monolicie modułowym, rozważmy tradycyjną architekturę warstwową (została opisana w rozdziale 10.). Jej komponenty są definiowane i organizowane według ich możliwości *technicznych*, czyli warstwy prezentacji, warstwy biznesowej, warstwy utrwalania itd. Na przykład logika prezentacji służąca do utrzymywania informacji o profilu klienta może być reprezentowana przez komponent z przestrzenią nazw *com.app.presentation.customer.profile*. Trzeci węzeł w przestrzeni odnosi się do *technicznego* zagadnienia warstwy (w tym przypadku jest to warstwa prezentacji).

I odwrotnie, komponenty monolitu modułowego są organizowane przeważnie z użyciem *domeny*. W związku z tym w architekturze monolitu modułowego ten sam komponent utrzymywania profilu klienta będzie reprezentowany przez przestrzeń nazw *com.app.customer.profile*. Tutaj trzeci węzeł przestrzeni nazw odnosi się do zagadnienia *domenowego*, a nie technicznego. Zależnie od złożoności komponentu przestrzeń nazw może być dalej dzielona według zagadnienia technicznego po wystąpieniu zagadnienia domenowego (na przykład *com.app.customer.profile .presentation* lub *com.app.customer.profile.business*).

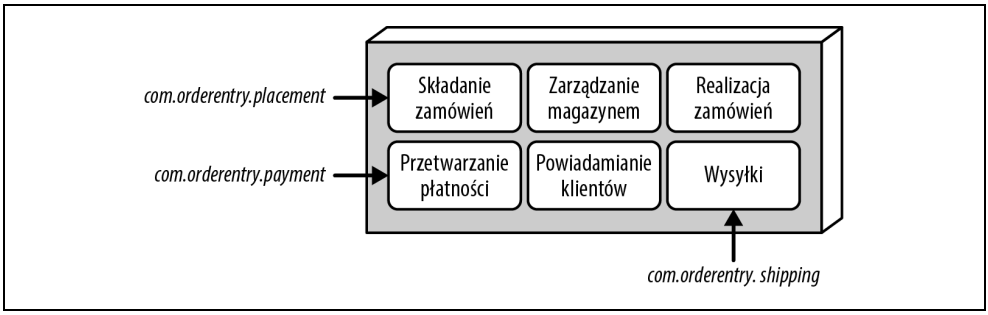
Szczegóły stylu

W tym stylu architektonicznym domeny (lub poddomeny w niektórych przypadkach) nazywa się **modułami**. Można je organizować na dwa sposoby. Najprostsza architektura to **struktura monolityczna**, w ramach której wszystkie moduły i odpowiadające im komponenty logiczne znajdują się w tej samej bazie kodu, opisane za pomocą obejmującej je przestrzeni nazw lub struktury katalogów. Nieco bardziej złożoną opcją jest **struktura modułowa**, w której każdy moduł jest reprezentowany jako niezależny i odrębny artefakt (np. plik JAR lub DLL), a moduły łączy się podczas wdrażania w postaci pojedynczej jednostki monolitycznej oprogramowania.

Jak to często bywa w architekturze oprogramowania, wybór pomiędzy tymi dwiema opcjami strukturalnymi zależy od wielu czynników i kompromisów. W dalszej części rozdziału omówiono obie opcje i związane z nimi kompromisy.

Struktura monolityczna

W strukturze monolitycznej wszystkie moduły reprezentujące system znajdują się w jednym repozytorium kodu źródłowego. Cały kod powiązany z każdym modułem jest wdrażany jako jedna jednostka w trakcie dostarczania lub publikowania oprogramowania. Ta opcja strukturalna została zilustrowana na rysunku 11.2. Każdy moduł jest reprezentowany przez osobny katalog wyższego poziomu zawierający komponenty i wszelkie poddomeny, które tworzą dany moduł.



Rysunek 11.2. Przykład opcji struktury monolitycznej

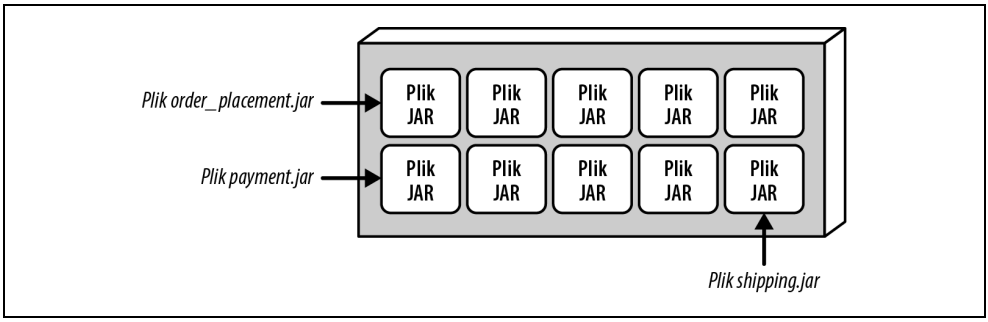
Następujące przestrzenie nazw ilustrują, jak może wyglądać monolit modułowy w przypadku architektury przedstawionej na rysunku 11.2:

```
com.orderentry.orderplacement
com.orderentry.inventorymanagement
com.orderentry.paymentprocessing
com.orderentry.notification
com.orderentry.fulfillment
com.orderentry.shipping
```

Jest to najprostsza opcja monolitu modułowego: cały kod źródłowy systemu znajduje się w jednym miejscu, co znacznie ułatwia jego utrzymanie, testowanie i wdrażanie. Aby jednak utrzymać granice każdego modułu, trzeba zapewnić ścisły nadzór (zajrzyj do podrozdziału „Nadzór" dalej w tym rozdziale). Choć ta opcja strukturalna jest prosta, projektanci mają tendencję do nadmiernego wykorzystywania kodu między modułami, a także do zbyt intensywnej komunikacji między nimi (zajrzyj do podrozdziału „Komunikacja między modułami"). Takie praktyki mogą zamienić dobrze zaprojektowany monolit modułowy w pozbawioną struktury bryły błotnej (ang. *big ball of mud*).

Struktura modułowa

W strukturze modułowej moduły są reprezentowane jako samodzielne artefakty (np. pliki JAR i DLL), a następnie łączone podczas wdrażania do postaci pojedynczej jednostki wdrożeniowej. Ta opcja z użyciem plików JAR na platformie języka Java jest zilustrowana na rysunku 11.3.



Rysunek 11.3. Przykład opcji struktury modułowej z użyciem plików JAR

Zaletą tej struktury jest to, że każdy moduł jest autonomiczny, co pozwala zespołom pracować nad oddzielnymi modułami (zajrzyj do podrozdziału „Kwestie dotyczące topologii zespołów"), a często nawet w ramach własnego, przeznaczonego do tych modułów repozytorium kodu źródłowego należącego do zespołu. Opcja ta sprawdza się dobrze, gdy moduły są w dużej mierze niezależne od innych modułów. Sprawdza się ona również w większych i bardziej złożonych systemach, w których każdy moduł wymaga innego rodzaju wiedzy biznesowej lub eksperckiej. W przypadku struktury modułowej projektanci są mniej skłonni do nadmiernego ponownego wykorzystywania kodu lub zbyt intensywnej komunikacji między modułami (zajrzyj do podrozdziału „Komunikacja między modułami"). Opcja ta pozwala zwykle zapewnić bardziej przejrzyste granice między modułami i lepszą ogólną separację zakresu odpowiedzialności.

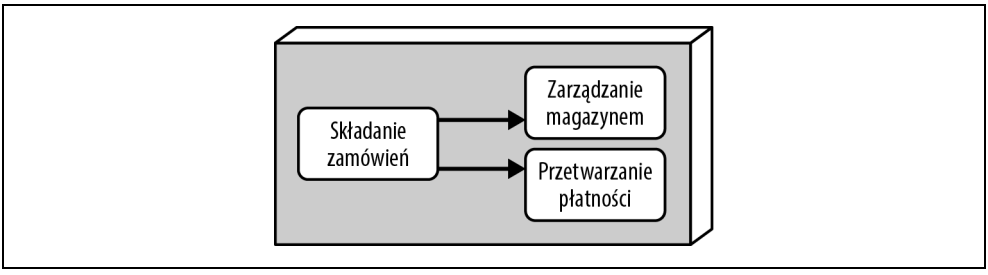
Ta opcja strukturalna traci jednak swoją skuteczność, gdy moduły zależne od siebie muszą się komunikować. W takiej sytuacji bardziej efektywne jest podejście oparte na strukturze monolitycznej.

Komunikacja między modułami

Komunikacja między modułami nigdy nie jest pożądana w ramach tego stylu architektonicznego, ale przyznajemy, że w wielu sytuacjach jest niezbędna. Na przykład w przypadku architektury przedstawionej na rysunku 11.2 moduł `OrderPlacement` musi komunikować się z modułem `InventoryManagement`, aby ten drugi dostosował stan magazynowy dla zamówionego produktu i przeprowadził dodatkowe przetwarzanie (na przykład aby zamówić więcej sztuk, jeśli zapasy są zbyt małe). Moduł `OrderPlacement` musi również komunikować się z modułem `PaymentProcessing` w celu uwzględnienia płatności za zamówienie. Istnieją dwie podstawowe opcje komunikacji między modułami, które omówiono w dwóch poniższych podpunktach.

Metoda bezpośrednia

Najprostsze rozwiązanie to bezpośrednia (równorzędna) komunikacja między modułami. W tym przypadku plik klasy w jednym module tworzy instancję klasy w innym module i wywołuje jej niezbędne metody, aby wykonać operację (rysunek 11.4).



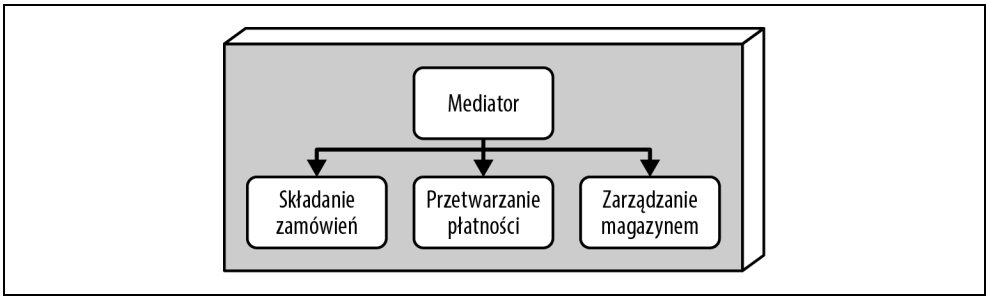
Rysunek 11.4. Bezpośrednia komunikacja między modułami

Problem z podejściem monolitycznym polega na tym, że projektanci mogą *zbyt* łatwo utworzyć instancję dowolnej klasy zawartej w innym module. Sprawia to, że łatwo jest przejść z architektury o odpowiedniej strukturze na antywzorzec bryły błotnej (rysunek 9.1).

Jednakże w przypadku struktury modułowej klasy zawarte w innym module mogą znajdować się w osobnych, zewnętrznych artefaktach (pliki JAR lub DLL), a nie w oddzielnym katalogu repozytorium kodu źródłowego. Moduł, który komunikuje się z innymi modułami, nie skompiluje się, dopóki nie uzyska odwołań do klas, co oznacza, że projektant musi utworzyć między tymi modułami zależność *fazy kompilacji*. Odpowiedzią na ten problem jest zwykle utworzenie wspólnej klasy interfejsu między modułami (w osobnym, współdzielonym pliku JAR lub DLL), aby każdy moduł mógł kompilować się niezależnie od innych. Tak czy inaczej, nadmierna komunikacja między modułami z wykorzystaniem struktury modułowej prowadzi do antywzorca „piekło" DLL (w przypadku platformy języka Java jest to antywzorzec „piekło" JAR; *https://pl.wikipedia.org/wiki/Piekło_DLL*).

Metoda mediatora

W ramach metody **mediatora** oddziela się od siebie moduły za pomocą komponentu mediatora w celu utworzenia warstwy abstrakcji między modułami. Mediator odgrywa rolę orkiestratora, przyjmując żądania i dostarczając je do odpowiednich modułów. Tę metodę zilustrowano na rysunku 11.5.



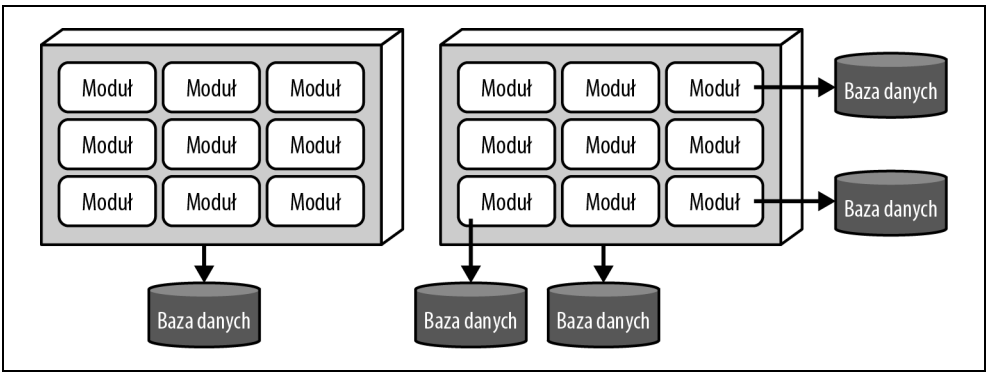
Rysunek 11.5. Mediator oddziela moduły, aby nie musiały się one komunikować ze sobą

Spostrzegawczy czytelnik zauważy, że choć metoda mediatora powoduje oddzielenie modułów, każdy z nich jest w efekcie sprzężony z mediatorem. Podejście to nie eliminuje *wszystkich* połączeń i zależności, ale upraszcza architekturę i utrzymuje moduły niezależne od siebie. Zauważ, że to *mediator*, a nie zależne moduły, potrzebuje pewnego rodzaju interfejsu API lub innego interfejsu do wywoływania funkcjonalności w innych modułach.

Topologie danych

Ponieważ architektura monolitu modułowego jest zazwyczaj wdrażana jako pojedyncza jednostka oprogramowania, opiera się przeważnie na topologii monolitycznej bazy danych. Korzystanie z jednej bazy danych pomaga zredukować ilość komunikacji między modułami, ponieważ dane są współużytkowane. Jeśli jednak moduły są niezależne od siebie i realizują

konkretne funkcje, mogą również mieć własne bazy danych zawierające dane kontekstowe nawet pomimo tego, że sama architektura jest monolityczna. Te dwie opcje topologii bazy danych są zilustrowane na rysunku 11.6.



Rysunek 11.6. Dane mogą być monolityczne albo moduły mogą mieć własne bazy danych

Kwestie dotyczące technologii chmury

Choć architektury monolitu modułowego można wdrażać w środowiskach chmury (zwłaszcza gdy system jest niewielki), generalnie nie są one dobrze dostosowane do wdrożeń w ramach tej technologii. Ich monolityczna natura sprawia, że rzadziej mogą korzystać z udostępniania zasobów na żądanie, jakie oferują środowiska chmury. Pomimo to mniejsze systemy zaimplementowane w ramach tego stylu architektonicznego nadal mogą korzystać z wielu usług chmury, takich jak magazyn plików, baza danych i przesyłanie komunikatów.

Typowe zagrożenia

Podobnie jak w przypadku każdego systemu monolitycznego, głównym elementem ryzyka związanym ze stylem architektonicznego monolitu modułowego jest to, że może on stać się zbyt duży, aby można było go właściwie utrzymywać, testować i wdrażać. Architektury monolityczne same w sobie nie są złe. Problemy zaczynają się pojawiać, gdy stają się one zbyt duże. To, co oznacza „zbyt duże", różni się w zależności od systemu, ale oto niektóre sygnały ostrzegawcze, że system może być zbyt duży:

- Wprowadzanie zmian trwa zbyt długo.
- Modyfikowanie jednego obszaru systemu powoduje, że inne obszary nieoczekiwanie przestają działać.
- Dokonując zmian, członkowie zespołu wchodzą sobie w drogę.
- Uruchomienie systemu trwa zbyt długo.

Innym elementem ryzyka jest nadmierne ponowne wykorzystanie kodu. Wielokrotne użycie kodu i udostępnianie go jest niezbędną częścią procesu tworzenia oprogramowania, ale w przypadku tego stylu architektonicznego zbyt duża skala ponownego stosowania kodu rozmywa granice między modułami, prowadząc architekturę do ryzykownej strefy *monolitu pozbawionego struktury*, czyli architektury monolitycznej z tak mocno wzajemnie powiązanym kodem, że nie można go rozdzielić.

Kolejnym elementem ryzyka w ramach tego stylu architektonicznego jest zbyt intensywna komunikacja między modułami. W idealnej sytuacji moduły powinny być niezależne i autonomiczne. Jak już wspomniano, normalne (a czasem konieczne) jest to, że niektóre moduły komunikują się ze sobą, a zwłaszcza w obrębie złożonego przepływu pracy. Jeśli jednak komunikacja między modułami jest przesadna, może to być symptom tego, że przede wszystkim domeny mogły zostać źle zdefiniowane. W takich przypadkach warto ponownie przemyśleć zmianę definicji domen w celu uwzględnienia złożonych przepływów pracy i wzajemnych zależności.

Nadzór

Głównym artefaktem w stylu architektonicznym monolitu modułowego jest *moduł*, który reprezentuje określoną domenę lub poddomenę, a ponadto ma zwykle postać struktury katalogów lub przestrzeni nazw (lub strukturę pakietów na platformie języka Java). Z tego powodu jedną z pierwszych form zautomatyzowanego nadzoru, jaką mogą zastosować architekci, jest zdefiniowanie i zapewnienie zgodności z modułami używanymi w architekturze.

Aby utworzyć kod sprawdzający w ramach zautomatyzowanego nadzoru, architekci korzystają z szeregu narzędzi, takich jak ArchUnit (*https://archunit.org/*) dla platformy języka Java, ArchUnitNet (*https://github.com/TNG/ArchUnitNET*), NetArchTest (*https://github.com/BenMorris/NetArchTest*) dla platformy .NET, PyTestArch (*https://pypi.org/project/PyTestArch/*) dla języka Python oraz TSArch (*https://github.com/ts-arch/ts-arch*) dla języków TypeScript i JavaScript. Pseudokod z listingu 11.1 zapewnia, że cały kod źródłowy przedstawiony w przykładzie architektury widocznym na rysunku 11.2 znajduje się w obrębie jednej z wymienionych przestrzeni nazw, które reprezentują każdy moduł zdefiniowany w systemie.

Listing 11.1. Pseudokod zapewniający zgodność kodu ze zdefiniowanymi modułami systemu

```
# Poniższe przestrzenie nazw reprezentują moduły w systemie
LIST module_list = {
   com.orderentry.orderplacement,
   com.orderentry.inventorymanagement,
   com.orderentry.paymentprocessing,
   com.orderentry.notification,
   com.orderentry.fulfillment,
   com.orderentry.shipping
   }
# Uzyskanie listy przestrzeni nazw w systemie
LIST namespace_list = get_all_namespaces(root_directory)
```

```
# Zapewnienie, że wszystkie nazwy przestrzeni zaczynają się
# od nazwy jednego z wymienionych modułów
FOREACH namespace IN namespace_list {
   IF NOT namespace.starts_with(module_list) {
      send_alert(namespace)
   }
}
```

Jeśli projektant utworzy jakiekolwiek dodatkowe przestrzenie nazw lub katalogi wyższego poziomu poza zdefiniowanymi modułami i ich odpowiednimi przestrzeniami nazw (lub katalogami), otrzyma alert wskazujący, że kod źródłowy nie jest zgodny z architekturą.

Ta forma nadzoru sprawdza się dobrze w przypadku opcji struktury monolitycznej (zajrzyj do punktu „Struktura monolityczna" wcześniej w tym rozdziale), ale jest bardziej problematyczna w przypadku opcji struktury modułowej (zajrzyj do wcześniejszego punktu „Struktura modułowa"), ponieważ kod może nie znajdować się w tym samym repozytorium kodu źródłowego. W strukturze modułowej każdy moduł trzeba testować osobno, co zaprezentowano w listingu 11.2.

Listing 11.2. Pseudokod sprawdzający poprawność modułu InventoryManagement

```
# Pobranie listy przestrzeni nazw w systemie
LIST namespace_list = get_all_namespaces(root_directory)

# Zapewnienie, że wszystkie nazwy przestrzeni zaczynają się od łańcucha com.orderentry.inventorymanagement
FOREACH namespace IN namespace_list {
   IF NOT namepace.starts_with("com.orderentry.inventorymanagement") {
      send_alert(namespace)
   }
}
```

Innym sposobem nadzorowania architektury monolitu modułowego jest kontrolowanie ilości komunikacji między modułami. Określenie, co oznacza „zbyt dużo" w przypadku komunikacji, jest bardzo subiektywne i zależy od systemu, ale przeważnie architekci powinni starać się minimalizować liczbę wzajemnych zależności między modułami. Listing 11.3 zawiera pseudokod zapewniający, że maksymalna, łączna liczba wzajemnych zależności nie przekracza limitu pięciu punktów komunikacji (lub sprzężenia).

Listing 11.3. Pseudokod ograniczający łączną liczbę zależności dowolnego modułu

```
# Przejście struktury katalogów oraz zebranie modułów i plików z kodem źródłowym
# zawartych w tych modułach
LIST module_list = {
    com.orderentry.orderplacement,
    com.orderentry.inventorymanagement,
    com.orderentry.paymentprocessing,
    com.orderentry.notification,
    com.orderentry.fulfillment,
    com.orderentry.shipping
    }

MAP module_source_file_map
FOREACH module IN module_list {
  LIST source_file_list = get_source_files(module)
  ADD module, source_file_list TO module_source_file_map
```

```
}

# Ustalenie, ile odwołań istnieje dla każdego pliku źródłowego, i wysłanie alertu,
# jeśli łączna liczba zależności w systemie przekroczy 5
FOREACH module, source_file_list IN module_source_file_map {
  FOREACH source_file IN source_file_list {
    incoming count = used_by_other_module(source_file, module_source_file_map) {
    outgoing_count = uses_other_module(source_file) {
    total_count = incoming count + outgoing count
  }
  IF total_count > 5 {
    send_alert(module, total_count)
  }
}
```

Ostatnią formą zautomatyzowanego nadzoru jest zapewnienie, że moduły pozostają niezależne od siebie, co polega na ograniczeniu możliwości komunikowania się jednego konkretnego modułu z innym. Na przykład na rysunku 11.2 moduł `OrderPlacement` nie powinien komunikować się z modułem `Shipping`. Listing 11.4 zawiera kod Java narzędzia ArchUnit, który nadzoruje tę zależność.

Listing 11.4. Kod Java narzędzia ArchUnit nadzorujący ograniczenia dotyczące zależności między konkretnymi modułami

```
public void order_placement_cannot_access_shipping() {
   noClasses().that()
   .resideInAPackage("..com.orderentry.orderplacement..")
   .should().accessClassesThat()
   .resideInAPackage("..com.orderentry.shipping..")
   .check(myClasses);
}
```

Kwestie dotyczące topologii zespołów

Ponieważ monolit modułowy uważa się za architekturę dzieloną według domen, najlepiej sprawdza się on wtedy, gdy zespoły również są dopasowywane na podstawie obszaru domenowego (np. zespoły wielofunkcyjne ze specjalizacją). Gdy pojawia się wymaganie oparte na domenie, członkowie zespołu wielofunkcyjnego ukierunkowanego na domenę mogą wspólnie zająć się tą kwestią, począwszy od logiki prezentacji, a skończywszy na bazie danych. Z kolei zespoły zorganizowane według kategorii technicznych (np. zespoły rozwijające interfejsy użytkownika, zespoły zaplecza, zespoły baz danych itd.) nie sprawdzają się dobrze w przypadku tego stylu architektonicznego, co wynika głównie z dzielenia ich według domeny. Przypisywanie wymagań domenowych zespołom zorganizowanym w sposób techniczny wymaga dużej ilości komunikacji i współpracy, co często okazuje się trudne.

Oto niektóre kwestie dotyczące dostosowywania do stylu monolitu modułowego konkretnych topologii zespołów, które opisano w rozdziale 9. w podrozdziale „Architektura i topologie zespołów”:

Zespoły skupione na strumieniu

Zespoły skupione na strumieniu odpowiadają generalnie za przepływ w obrębie systemu od początku do końca, co dobrze współgra z monolitycznym i ogólnie autonomicznym charakterem monolitu modułowego.

Zespoły wspierające

Ze względu na wysoki poziom modułowości i separacji zakresu odpowiedzialności w przypadku tego stylu topologia zespołów wspierających również dobrze się sprawdza. Specjaliści i członkowie zespołów o kompetencjach przekrojowych mogą przedstawiać sugestie i przeprowadzać eksperymenty, wprowadzając do systemu dodatkowe moduły z minimalnym wpływem na inne istniejące moduły.

Zespoły zajmujące się skomplikowanymi podsystemami

Każdy moduł w architekturze monolitu modułowego generalnie pełni określoną funkcję zależną od jego domeny lub poddomeny (np. `PaymentProcessing`). Sprawdza się to dobrze w przypadku topologii zespołów zajmujących się skomplikowanymi podsystemami, ponieważ różni członkowie zespołu mogą skupić się na złożonym przetwarzaniu w ramach domeny lub poddomeny niezależnie od innych członków (i modułów).

Zespoły platformowe

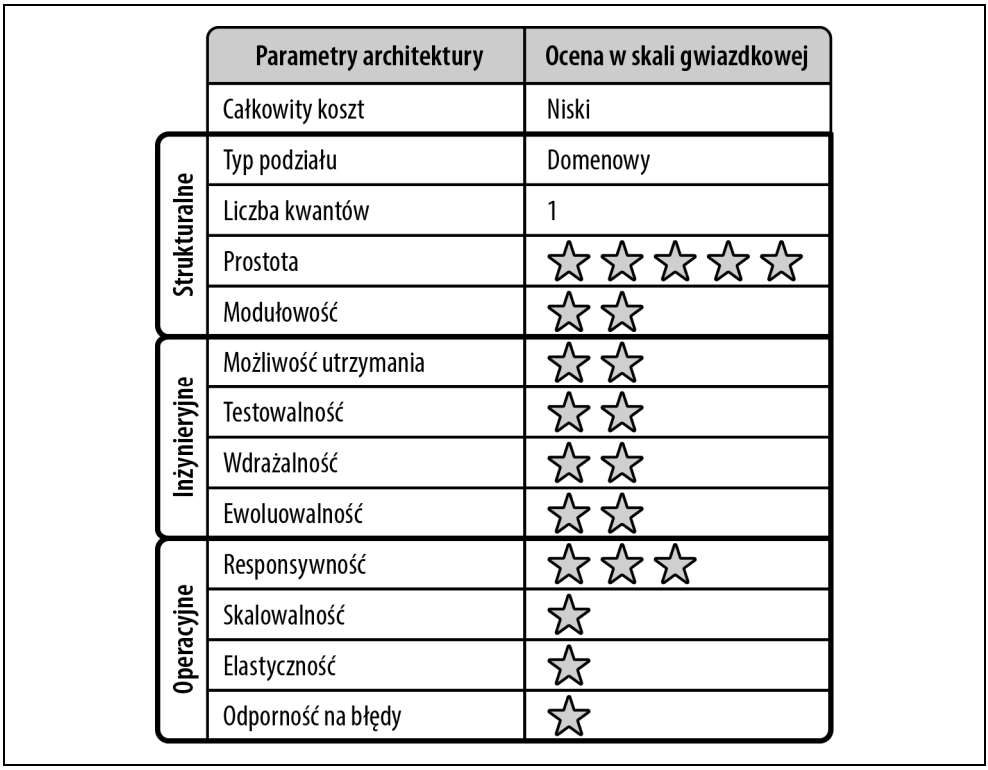
Projektanci mogą skorzystać z zalet topologii zespołów platformowych, używając wspólnych narzędzi, usług, interfejsów API i zadań, głównie ze względu na wysoki stopień modułowości w tym stylu architektonicznym.

Parametry stylu

Ocena jednogwiazdkowa w tabeli ocen parametrów (rysunek 11.7) oznacza, że określony parametr architektoniczny nie jest dobrze obsługiwany w ramach architektury, natomiast ocena pięciogwiazdkowa wskazuje, że jej parametr jest jednym z najsilniejszych elementów stylu architektonicznego. Parametry zawarte w tabeli opisano i zdefiniowano w rozdziale 4.

Styl architektoniczny monolitu modułowego jest architekturą *dzieloną według domen*, gdyż jej logikę aplikacji rozbija się na moduły. Jako że architekturę zazwyczaj implementuje się jako monolityczną jednostkę wdrożeniową, jej kwant architektury wynosi zwykle 1.

Główne zalety stylu architektonicznego monolitu modułowego to ogólne koszty, prostota i modułowość. Ze względu na swoją monolityczną naturę architektury te nie mają złożoności charakterystycznej dla stylów architektury rozproszonej. Są one prostsze i łatwiejsze do zrozumienia, a ponadto cechują się względnie niewielkim kosztem budowy i utrzymania. Modułowość architektoniczną osiąga się w wyniku rozdzielenia zakresu odpowiedzialności między różnymi modułami reprezentującymi domeny i poddomeny.



	Parametry architektury	Ocena w skali gwiazdkowej
	Całkowity koszt	Niski
Strukturalne	Typ podziału	Domenowy
	Liczba kwantów	1
	Prostota	★★★★★
	Modułowość	★★
Inżynieryjne	Możliwość utrzymania	★★
	Testowalność	★★
	Wdrażalność	★★
	Ewoluowalność	★★
Operacyjne	Responsywność	★★★
	Skalowalność	★
	Elastyczność	★
	Odporność na błędy	★

Rysunek 11.7. Oceny (gwiazdki) parametrów architektonicznych monolitu modułowego

Możliwości wdrażania i testowania, mimo że ocenione jedynie na dwie gwiazdki, w monolicie modułowym ze względu na swój poziom modułowości plasują się nieco wyżej niż w architekturze warstwowej. Niemniej jednak ten styl architektoniczny nadal jest monolitem, a zatem negatywny wpływ na te oceny ma rytuał, ryzyko, częstotliwość wdrożeń i kompletność testów.

Elastyczność i skalowalność architektury monolitu modułowego są oceniane bardzo nisko (jedna gwiazdka) głównie z powodu wdrożeń monolitycznych. Choć można sprawić, aby niektóre funkcje w monolicie były bardziej skalowalne niż inne, wymaga to zwykle bardzo złożonych technik projektowych (takich jak wielowątkowość, wewnętrzne przesyłanie komunikatów oraz inne praktyki z zakresu przetwarzania równoległego), do których ta architektura nie jest dobrze przystosowana.

Wdrożenia architektury monolitu modułowego nie zapewniają tolerancji na błędy. Jeśli jedna niewielka część architektury spowoduje błąd braku pamięci, cała jednostka aplikacji ulegnie awarii. Co więcej, jak w większości aplikacji monolitycznych, ogólna dostępność jest ograniczona przez dużą wartość wskaźnika średniego przywracania po awarii (MTTR — *Mean Time to Recovery*), a czasy uruchamiania zwykle mierzy się w minutach.

Kiedy stosować?

Ze względu na swoją prostotę i niski koszt styl architektoniczny monolitu modułowego to właściwa opcja w sytuacjach, gdy ma się do czynienia z ograniczonym budżetem i napiętymi terminami. Jest to również dobry wybór w początkach pracy z nowym systemem. Jeśli kierunek architektoniczny systemu nie jest jeszcze jasny, często bardziej efektywne będzie rozpoczęcie od monolitu modułowego i późniejsze przejście na bardziej skomplikowany oraz kosztowny styl architektury rozproszonej, takiej jak architektura oparta na usługach (zajrzyj do rozdziału 14.) lub mikrousługi (zobacz rozdział 18.), niż skorzystanie od razu z architektury rozproszonej.

Monolit modułowy jest również dobrą propozycją w przypadku zespołów ukierunkowanych na domeny, takich jak wyspecjalizowane zespoły wielofunkcyjne. Pozwala to każdemu zespołowi skupić się od początku do końca na konkretnym module w obrębie architektury przy minimalnym stopniu koordynacji z innymi zespołami domenowymi. Ten styl architektoniczny dobrze sprawdza się także w sytuacjach, w których większość zmian w systemie opiera się na domenie (np. dodawanie dat ważności do pozycji znajdujących się na liście życzeń klienta).

I wreszcie, ponieważ monolit modułowy to architektura dzielona według domen, jest odpowiedni dla zespołów stosujących metodykę DDD (*https://pl.wikipedia.org/wiki/Domain-driven_design*).

Kiedy nie stosować?

Głównym powodem, dla którego nie należy stosować tego stylu architektonicznego, jest sytuacja, w której systemy lub produkty wymagają wysokich poziomów pewnych parametrów operacyjnych, takich jak skalowalność, elastyczność, dostępność, tolerancja na błędy, responsywność i wydajność. Podobnie do większości architektur monolitycznych, monolit modułowy nie pozwala ich zapewnić.

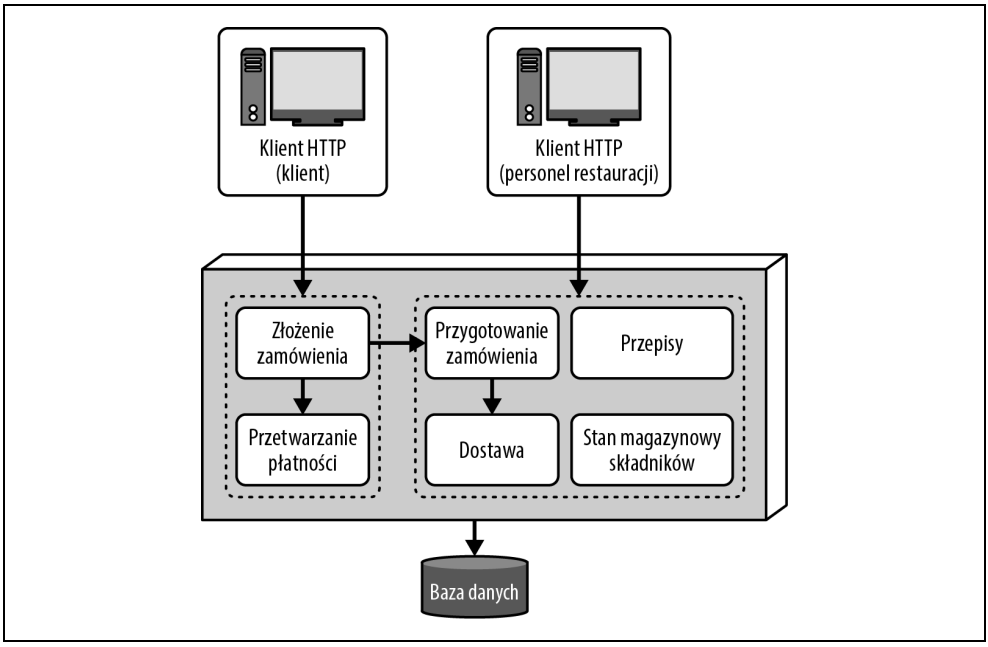
Unikaj stosowania monolitu modułowego, gdy większość zmian ma charakter techniczny (np. ciągła wymiana interfejsu użytkownika lub technologii bazy danych). Ponieważ tę architekturę dzieli się według domen, takie zmiany wpływają na każdy moduł i wymagają zwykle znacznej ilości komunikacji i koordynacji między zespołami domenowymi. W takich sytuacjach styl architektury warstwowej (zajrzyj do rozdziału 10.) jest znacznie lepszym wyborem.

Przykłady i przypadki użycia

EasyMeals to nowa lokalna restauracja oferująca dostawę pracującym osobom, które nie zawsze mają czas na gotowanie po powrocie do domu po intensywnym dniu. Głodni klienci mogą zamówić za pośrednictwem sieci smaczny obiad i otrzymać go pod drzwi w ciągu godziny.

Jako niewielka, lokalna restauracja, EasyMeals nie ma wysokich wymagań dotyczących skalowalności ani responsywności. Ponieważ jej budżet jest ograniczony, jej właściciele nie zamierzają wydawać wiele pieniędzy na rozbudowany system oprogramowania. Postać tego problemu biznesowego sprawia, że monolit modułowy to dla restauracji EasyMeals dobra propozycja.

Na rysunku 11.8 pokazano, jak może wyglądać prosty system zarządzania restauracją Easy-Meals oparty na stylu architektonicznym monolitu modułowego.



Rysunek 11.8. System zamówień i zarządzania niewielką restauracją oparty na stylu architektonicznym monolitu modułowego

Klienci uzyskują dostęp do modułów `PlaceOrder` i `PaymentProcessing` za pośrednictwem specjalnego interfejsu użytkownika korzystającego z kolejnego tego rodzaju interfejsu. Poszczególne moduły tego systemu są reprezentowane przez następujące przestrzenie nazw:

```
com.easymeals.placeorder
com.easymeals.payment
com.easymeals.prepareorder
com.easymeals.delivery
com.easymeals.recipes
com.easymeals.inventory
```

Moduł `PlaceOrder` pozwala każdemu klientowi wyświetlić menu, wybrać dania, dodać swoje personalia, adres i informacje o płatności, a następnie złożyć zamówienie. Komponenty tego modułu mogą być reprezentowane przez następujące przestrzenie nazw, które obejmują kod źródłowy implementujący każdą z głównych funkcji:

```
com.easymeals.placeorder.menu
com.easymeals.placeorder.shoppingcart
com.easymeals.placeorder.customerdata
com.easymeals.placeorder.paymentdata
com.easymeals.placeorder.checkout
```

Przykład ten ilustruje to, że moduł w monolicie modułowym składa się z co najmniej jednego *komponentu* (rozdział 8.).

Moduł `PaymentProcessing` jest odpowiedzialny za obsługę płatności. Restauracja EasyMeals akceptuje karty kredytowe, karty debetowe i usługę PayPal. Modułowość tej architektury sprawia, że łatwo można dodać kolejny typ płatności (np. punkty lojalnościowe). Klienci wprowadzają te informacje w module `PlaceOrder`, który przekazuje je do modułu `PaymentProcessing`. Komponenty tego modułu mogą być reprezentowane przez następujące przestrzenie nazw:

```
com.easymeals.payment.creditcard
com.easymeals.payment.debitcard
com.easymeals.payment.paypal
```

Po dokonaniu płatności za zamówienie moduł `PlaceOrder` komunikuje się z modułem `PrepareOrder`, który wyświetla całe zamówienie dla personelu kuchennego. Po przygotowaniu posiłku personel oznacza zamówienie jako gotowe do dostawy, a następnie jest ono przesyłane do modułu `Delivery`. Komponenty wewnątrz modułu `PrepareOrder` są reprezentowane przez następujące przestrzenie nazw:

```
com.easymeals.prepareorder.displayorder
com.easymeals.prepareorder.ready
```

Moduł `Delivery` przydziela zamówienie dostawcy i wskazuje adres dostawy. Pozwala on dostawcy oznaczyć zamówienie jako dostarczone, co kończy cykl istnienia tego konkretnego zamówienia, a także rejestrować wszelkie problemy (np. agresywny pies przy bramie wejściowej lub klient, którego nie zastano w domu). Komponenty wewnątrz modułu `Delivery` są reprezentowane przez następujące przestrzenie nazw:

```
com.easymeals.delivery.assign
com.easymeals.delivery.issues
com.easymeals.delivery.complete
```

Moduł `Recipes` umożliwia kucharzom i personelowi zarządzającemu dodawanie pozycji do menu oraz utrzymywanie listy składników i wymiarów dla każdego dania w menu. Komponenty w obrębie modułu `Recipes` są reprezentowane przez następujące przestrzenie nazw:

```
com.easymeals.recipes.view
com.easymeals.recipes.maintenance
```

I wreszcie moduł `IngredientsInventory` zapewnia, że dostępna jest wystarczająca ilość składników dań w menu. Moduł jest nieco bardziej złożony niż pozostałe. Zawiera on zaawansowany komponent sztucznej inteligencji, który prognozuje wolumen sprzedaży, aby zautomatyzować proces zapewniania składników na cały tydzień.

Następujące przestrzenie nazw reprezentują komponenty wewnątrz modułu `Ingredients ↪Inventory`:

```
com.easymeals.inventory.maintenance
com.easymeals.inventory.forecasting
com.easymeals.inventory.ordering
com.easymeals.inventory.suppliers
com.easymeals.inventory.invoices
```

I to wszystko! Prostota i poziom modułowości monolitu modułowego sprawiają, że stosunkowo łatwo można zlokalizować kod i utrzymywać go, na przykład naprawić błąd lub dodać nową opcję. Ilustruje to siłę tego prostego i przejrzystego stylu architektonicznego.

Architektura oprogramowania to dziś nie tylko kwestia doświadczenia czy intuicji — staje się pełnoprawną dziedziną inżynieryjną, która zakłada powtarzalność, rygor i skuteczną analizę.

Ta książka pokazuje, jak projektować systemy w sposób świadomy, oparty na sprawdzonych zasadach i analizie kompromisów. Zawiera rozdziały poświęcone najnowszym spostrzeżeniom związanym z tą dziedziną. Obejmuje zarówno klasyczne tematy (na przykład wzorce architektoniczne, wyodrębnianie komponentów, dokumentowanie architektury), jak i nowe zagadnienia, takie jak architektura ewolucyjna, wpływ AI na projektowanie systemów czy organizacja zespołów developerskich. Dokładnie wyjaśnia zasady, które mogą być zastosowane do wszystkich zestawów rozwiązań technologicznych. W książce duży nacisk położono na analizę kompromisów, która pozwala na obiektywną ocenę rozwiązań technologicznych. Architektura oprogramowania została tu ujęta jako dziedzina inżynieryjna — z całym dorobkiem ostatniej dekady.

W książce znajdziesz wiedzę niezbędną do tego by zgłębić nowoczesną architekturę oprogramowania z perspektywy współczesnych realiów

Raju Gandhi, współautor *Architektury oprogramowania. Rusz głową!*

W książce:

- style i wzorce architektoniczne
- kluczowe składniki i cechy nowoczesnych projektów
- umiejętności miękkie pomocne w pracy architekta
- nowoczesne praktyki inżynierii oprogramowania
- architektura jako dziedzina inżynieryjna

Mark Richards jest doświadczonym architektem oprogramowania i autorem specjalistycznych książek. Zajmuje się projektowaniem i wdrażaniem mikrousług, a także innych systemów o architekturze rozproszonej.

Neal Ford jest uznanym ekspertem z zakresu tworzenia oprogramowania. Pełni funkcję dyrektora w ThoughtWorks. Pisze książki i artykuły, tworzy również materiały dydaktyczne poświęcone architekturze oprogramowania.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2996-8
9 788328 929968
Cena: 129,00 zł