Vaughn **Vernon**

# DDD

## dla architektów oprogramowania

Helion

Tytuł oryginału: Implementing Domain-Driven Design

Tłumaczenie: Radosław Meryk

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-2547-0



Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*ftp://ftp.helion.pl/przyklady/dddaro.zip*

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

Printed in Poland.

# Spis treści

# Rozdział 1.

# Wprowadzenie w DDD

*Projekt nie jest tylko tym, jak rzecz wygląda i jakie sprawia wrażenie. Projekt opisuje to, jak ona działa.*

*— Steve Jobs*

Pracując nad oprogramowaniem, staramy się, by było ono wysokiej jakości. Pewną jakość osiągamy poprzez zastosowanie testów, które pomagają nam dostarczać oprogramowanie bez olbrzymiej liczby błędów. Jednakże, nawet jeśli moglibyśmy stworzyć oprogramowanie zupełnie wolne od błędów, samo to niekoniecznie oznacza, że zaprojektowany model oprogramowania jest wysokiej jakości. Model oprogramowania — *sposób*, w jaki oprogramowanie wyraża rozwiązanie poszukiwanego problemu biznesowego — w dalszym ciągu może mieć istotne wady. Dostarczanie oprogramowania z małą liczbą defektów jest oczywiście dobre. Mimo to możemy sięgnąć wyżej — po dobrze zaprojektowany model oprogramowania, który jawnie odzwierciedla zamierzony cel biznesowy — a nasza praca może nawet osiągnąć poziom *doskonały*.

Podejście rozwoju oprogramowania określane jako *Domain-Driven Design*, albo DDD, istnieje po to, by było nam łatwiej odnieść sukces w tworzeniu wysokiej jakości projektów modelu oprogramowania. Jeśli metodyka DDD zostanie prawidłowo zaimplementowana, pomaga osiągnąć punkt, w którym *nasz projekt dokładnie prezentuje sposób, w jaki działa oprogramowanie*. Celem tej książki jest udzielenie czytelnikom pomocy w prawidłowej implementacji DDD.

Może DDD jest dla Ciebie zupełną nowością. Być może już próbowałeś posłużyć się DDD i przychodziło Ci to z trudem, a może wcześniej skutecznie zastosowałeś tę metodykę. Niezależnie od okoliczności bez wątpienia czytasz tę książkę dlatego, że chcesz poprawić swoje umiejętności implementacji DDD — i możesz to osiągnąć. „Mapa drogowa" rozdziału pomoże Ci zaspokoić Twoje specyficzne potrzeby.

**Mapa drogowa rozdziału**

- Podczas zmagań ze złożonością sprawdzisz, co Twoim projektom i zespołom może zaoferować DDD.
- Dowiesz się, jak ocenić projekt, by się przekonać, czy zasługuje on na zainwestowanie w DDD.

- Rozważysz popularne alternatywy dla DDD i zastanowisz się, dlaczego ich stosowanie często prowadzi do problemów.
- Nauczysz się podstaw DDD przy okazji zapoznawania się z pierwszymi krokami w projekcie, które powinieneś zrobić.
- Dowiesz się, jak przekazać zalety metodyki DDD kierownictwu, ekspertom dziedziny i technicznym uczestnikom projektu.
- Staniesz wobec wyzwań stosowania DDD uzbrojony w wiedzę o tym, jak możesz odnieść sukces.
- Przyjrzysz się zespołowi, który uczy się sposobów implementacji DDD.

Czego powinieneś oczekiwać od DDD? Nie jest to trudny, skomplikowany, ceremonialny proces, który blokuje drogę do postępów. Zamiast tego oczekuj stosowania zwinnych technik projektowania, którym prawdopodobnie już zaufałeś. Poza tym przygotuj się na poznanie metod, które pomagają zyskać głęboki wgląd w Twoją dziedzinę biznesową. Jednocześnie dają perspektywę stworzenia modeli oprogramowania, które są testowalne, elastyczne, uważnie zorganizowane i starannie wykonane. Są po prostu wysokiej jakości.

DDD daje nam narzędzia zarówno do *modelowania strategicznego, jak i taktycznego*. Narzędzia te są konieczne do projektowania wysokiej jakości oprogramowania, które spełnia podstawowe cele biznesowe.

## Czy mogę zastosować DDD?

Możesz zastosować DDD, jeżeli masz:

- pasję do tworzenia doskonałego oprogramowania codziennie oraz upór, by osiągnąć ten cel;
- gorliwość do nauki i osiągania postępów oraz hart ducha, aby przyznać się do tego, że jest Ci to potrzebne;
- zdolności do rozumienia wzorców oprogramowania oraz sposobów ich prawidłowego stosowania;
- umiejętności i cierpliwość do odkrywania alternatywnych projektów z wykorzystaniem sprawdzonych zwinnych metod wytwarzania oprogramowania;
- odwagę do stawiania wyzwań stanowi istniejącemu;
- pragnienie zwracania uwagi na szczegóły i umiejętność ich dostrzegania oraz upodobanie do eksperymentowania i odkrywania;
- motywację do szukania sposobów na lepsze i bardziej inteligentne kodowanie.

Nie chcę powiedzieć, że krzywa nauki nie istnieje. Mówiąc otwarcie, krzywa nauki bywa stroma. Pomimo to niniejsza książka powstała, aby pomóc spłaszczyć tę krzywą tak bardzo, jak to możliwe. Moim celem jest udzielenie czytelnikowi i jego zespołowi pomocy w zastosowaniu metodyki DDD oraz zmaksymalizowaniu szans na odniesienie sukcesu.

DDD przede wszystkim nie dotyczy technologii. Podstawowe zasady DDD to: dyskusja, słuchanie, zrozumienie, odkrywanie i wartość biznesowa — wszystko po to, aby scentralizować wiedzę. Jeżeli masz zdolność *rozumienia biznesu*, w którym działa Twoje przedsiębiorstwo, to zgodnie z planem minimum możesz uczestniczyć w procesie odkrywania modelu oprogramowania w celu stworzenia Języka Wszechobecnego. Oczywiście będziesz musiał nauczyć się o biznesie więcej — znacznie więcej. Mimo to jesteś na dobrej drodze do odniesienia sukcesu z DDD, ponieważ potrafisz rozumieć pojęcia swojego biznesu podczas rozwijania oprogramowania, a to tworzy właściwą podstawę do zastosowania DDD w całym procesie.

Czy posiadanie wieloletniego doświadczenia w rozwoju oprogramowania jest pomocne? Być może. Niemniej jednak doświadczenie w rozwijaniu oprogramowania nie daje zdolności słuchania i uczenia się od ekspertów dziedziny — ludzi, którzy wiedzą najwięcej na temat obszarów biznesu o wysokim priorytecie. Najwięcej można skorzystać z rozmów z osobami, które rzadko używają technicznego żargonu (lub nigdy tego nie robią). Trzeba nauczyć się uważnie ich słuchać. Trzeba uszanować ich punkt widzenia i zaufać, że znają temat znacznie lepiej niż my sami.

**Zaangażowanie ekspertów dziedziny przynosi duże korzyści**

Można dużo skorzystać na rozmowach z osobami, które rzadko używają technicznego żargonu (lub nigdy tego nie robią). Nie tylko Ty będziesz uczył się od nich. Istnieje duże prawdopodobieństwo, że oni także będą uczyli się od Ciebie.

Jedną z bardziej wartościowych cech DDD jest to, że eksperci dziedziny również *muszą posłuchać Ciebie*. Jesteście członkami tego samego zespołu. Chociaż może się to wydawać dziwne, eksperci dziedziny nie wiedzą wszystkiego o swoim biznesie i oni też powinni dowiedzieć się o nim więcej. Nie tylko Ty będziesz się uczył od nich. Istnieje duże prawdopodobieństwo, że oni także będą uczyli się od Ciebie. Twoje pytania o to, co oni wiedzą, najprawdopodobniej spowodują również odkrycie tych obszarów, które nie są im znane tak samo dobrze. Będziesz bezpośrednio zaangażowany w pomoc wszystkim osobom w zespole w zdobywaniu głębszego zrozumienia biznesu — można nawet powiedzieć: *kształtowania biznesu*.

To wspaniale, kiedy zespół uczy się i rozwija razem. Jeżeli dasz temu szansę, metodyka DDD to umożliwi.

**Nie mamy ekspertów dziedziny**

Ekspert dziedziny to nie jest tytuł zawodowy. Ekspertami dziedziny są ludzie, którzy bardzo dobrze znają obszar biznesu, w którym pracujesz. Często mają oni obszerną wiedzę w dziedzinie biznesowej. Zazwyczaj są to projektanci produktu albo osoby z działu sprzedaży.

Nie zwracaj uwagi na nazwy stanowisk. Poszukiwani przez Ciebie ludzie o dziedzinie, w której pracujesz, wiedzą więcej niż ktokolwiek inny i na pewno znacznie więcej niż Ty. *Znajdź ich. Posłuchaj. Naucz się czegoś od nich. Uwzględnij ich zdanie w projekcie kodu.*

Dotychczas osiągnęliśmy gotowość do spokojnego startu. Nie chciałbym jednak powiedzieć, że umiejętności techniczne są nieważne — że są czymś, bez czego można się obyć. Trzeba się zapoznać z zaawansowanymi pojęciami z zakresu *modelowania dziedziny* w oprogramowaniu. Nie oznacza to jednak, że będziemy zmuszeni robić coś, co jest ponad nasze siły. Jeśli Twoje umiejętności mieszczą się gdzieś pomiędzy wiedzą z książki *Head First Design Patterns* [Freeman et al.] a oryginalną zawartością pozycji *Design Patterns* [Gamma et al.] lub jeśli znasz bardziej zaawansowane wzorce, to masz naprawdę duże szanse na odniesienie sukcesu z DDD. Możesz na to liczyć. Będę robił wszystko, co w mojej mocy, aby stało się to możliwe. Będę się starał „obniżać poprzeczkę", niezależnie od tego, jaki jest wyjściowy poziom doświadczenia czytelnika.

**Co to jest Model Dziedziny?**

To model oprogramowania bardzo określonej dziedziny biznesowej, w której pracujesz. Często jest on zaimplementowany jako model obiektowy, w którym obiekty zawierają zarówno dane, jak i zachowania w harmonii z dokładnym znaczeniem biznesowym.

Stworzenie unikatowego, uważnie zaprojektowanego Modelu Dziedziny, stanowiącego sedno podstawowej, strategicznej aplikacji albo podsystemu, ma zasadnicze znaczenie dla stosowania metodologii DDD. Dzięki użyciu DDD Modele Dziedziny będą mniej rozbudowane i bardzo skoncentrowane. Stosując DDD, nigdy nie próbujemy zamodelować całego przedsiębiorstwa biznesowego w pojedynczym, olbrzymim Modelu Dziedziny. To jest dobre!

Warto rozważyć wymienione poniżej perspektywy osób, które mogą skorzystać na stosowaniu DDD. Wiem, że czytelnik pasuje do jednej z tych grup.

- *Nowicjusz, młodszy deweloper:* „Jestem młody, ze świeżymi pomysłami. Mam mnóstwo energii do kodowania i chcę mieć wpływ na projekt. Zdenerwował mnie jeden z projektów, w którym brałem udział. Nie oczekiwałem, że ten mój pierwszy »koncert« poza kampusem studenckim będzie oznaczał przerzucanie szuflą danych w tę i z powrotem, przy wykorzystaniu wielu prawie identycznych, a jednak redundantnych obiektów. Dlaczego ta architektura jest tak złożona, skoro w oprogramowaniu dzieje się tak niewiele? Dlaczego

tak jest? Kiedy staram się modyfikować kod, ulega on częstym awariom. Czy ktoś właściwie rozumie, co on powinien robić? Mam teraz kilka złożonych nowych funkcji do dodania. Regularnie tworzę *adaptery* wokół istniejących klas, dążąc do tego, by uniknąć kiczu. *Żadna przyjemność*. Jestem pewien, że jest coś, co mogę zrobić poza kodowaniem i debugowaniem dzień i noc tylko po to, aby dokończyć iteracje. Cokolwiek to jest, będę się starał to znaleźć i poznać. Usłyszałem rozmowy kilku osób na temat DDD. *To brzmi jak Gang Czterech, ale dostosowany do modelu dziedziny*. Doskonale".

Muszę się z tym zapoznać.

- ***Średnio zaawansowany deweloper:*** „Kilka ostatnich miesięcy spędziłem przy nowym projekcie. Moją rolą jest stworzenie różnicy. Rozumiem, co się dzieje w projekcie, ale brakuje mi głębokiej wiedzy, kiedy spotykam się ze starszymi deweloperami. Wydaje mi się, że niektóre sprawy przebiegają źle, ale nie jestem pewien, dlaczego tak jest. Mam zamiar pomóc zmienić sposób, w jaki są realizowane funkcje. Wiem, że zastosowanie określonej technologii do rozwiązania problemu pozwala dotrzeć tylko do pewnego miejsca. Ogólnie rzecz biorąc, to nie jest wystarczająco daleko. Potrzebuję *zdrowej techniki rozwoju oprogramowania*, która pomoże mi stać się mądrym i doświadczonym twórcą oprogramowania. Jeden ze starszych architektów, nowa osoba w projekcie, wspomniał o metodologii, która nazywa się DDD. Postanowiłem się temu przysłuchać".

  W tym momencie mówisz już jak doświadczony programista. Kontynuuj lekturę. Twoje nastawienie i „myślenie do przodu" będą nagrodzone.

- ***Starszy deweloper, architekt:*** „Używałem metodyki DDD w kilku projektach, ale odkąd objąłem nowe stanowisko, jeszcze jej nie stosowałem. Lubię możliwości *wzorców taktycznych*, lecz mógłbym osiągnąć znacznie więcej, wykorzystując *projektowanie strategiczne*. Kiedy czytałem [Evans], największe wrażenie wywarł na mnie Język Wszechobecny. *To potężne narzędzie*. Dyskutowałem z kilkoma kolegami z mojego zespołu i z kierownictwa, starając się wpłynąć na ich pogląd na temat zastosowania DDD. Perspektywy użycia tej metodyki budzą ogromne podekscytowanie jednego z nowych członków zespołu, a także kilku średnio zaawansowanych i starszych programistów. Kierownictwo nie podziela tego entuzjazmu. Dołączyłem do tej firmy niedawno i chociaż otrzymałem zadanie bycia liderem, to wydaje się, że organizacja jest zainteresowana wprowadzaniem gwałtownych usprawnień w mniejszym stopniu, niż myślałem. Wszystko jedno. I tak nie rezygnuję. Jeśli uzyskam poparcie innych deweloperów, to wiem, że będziemy w stanie razem to osiągnąć. Korzyści będą znacznie większe niż oczekiwania. Zaprosimy ludzi zajmujących się wyłącznie biznesem — ekspertów dziedziny — do bliższej współpracy z zespołami technicznymi *i rzeczywiście zainwestujemy w nasze rozwiązania*, zamiast ograniczać się wyłącznie do ich produkowania — iteracja po iteracji".

To jest dokładnie *to*, co powinien robić lider. W tej książce zamieszczono mnóstwo wskazówek, które pokazują, jak odnieść sukces, stosując *projektowanie strategiczne*.

- ***Ekspert dziedziny:*** „Uczestniczę w specyfikowaniu rozwiązań IT naszych biznesowych wyzwań już od długiego czasu. Być może oczekuję zbyt wiele, ale chciałbym, żeby deweloperzy lepiej rozumieli, co my tutaj robimy. Zawsze rozmawiają z nami tak, jakbyśmy byli nierozgarnięci. Nie rozumieją, że gdyby nie było nas, to oni nie mieliby nic do roboty ze swoimi komputerami. Deweloperzy zawsze w jakiś dziwny sposób mówią o tym, co robi nasze oprogramowanie. Jeżeli mówimy o A, oni mówią, że to tak naprawdę nazywa się B. *Wydaje mi się, że powinniśmy mieć jakiś rodzaj słownika lub mapy drogowej zawsze wtedy, gdy chcemy powiedzieć, czego potrzebujemy.* Jeśli nie pozwalamy deweloperom działać po swojemu, tzn. używać nazwy B w odniesieniu do czegoś, co my znamy jako A, to przestają współpracować. Tracimy w ten sposób mnóstwo czasu. *Dlaczego oprogramowanie nie może działać dokładnie tak, jak prawdziwi eksperci myślą o biznesie?*”.

  Dobrze to ująłeś. Jednym z największych problemów jest fałszywa potrzeba tłumaczenia przekazu pomiędzy ludźmi biznesu a pracownikami technicznymi. Ten rozdział jest dla Ciebie. Jak się przekonasz, *DDD stawia ekspertów dziedziny i deweloperów na tym samym poziomie*.

  I jeszcze niespodzianka! Niektórzy deweloperzy już skłaniają się w Twoją stronę. Pomóż im w tym.

- ***Menedżer:*** „Dostarczamy oprogramowanie. Efekty nie zawsze są zgodne z oczekiwaniami, a wprowadzanie zmian zwykle zajmuje więcej czasu, niż powinno. Deweloperzy stale mówią coś o jakiejś dziedzinie. Nie jestem pewien, czy powinniśmy koncentrować się na kolejnej technice albo metodologii, sądząc, że będzie ona rodzajem srebrnej kuli. Słyszałem to wcześniej już tysiąc razy. Próbujemy, entuzjazm zanika i znów wracamy do starego. Ciągle powtarzam, że powinniśmy trzymać kurs i przestać marzyć, ale zespół mnie naciska. Pracują ciężko, więc powinienem ich wysłuchać. *Są inteligentnymi ludźmi i zasługują na szansę usprawnienia rzeczywistości*. Jeśli jej im nie dam, zdenerwują się i odejdą. Mógłbym im dać trochę czasu na naukę i dostrojenie się do nowej techniki, gdybym dostał zielone światło z wyższego szczebla zarządzania. Myślę, że mógłbym zyskać tę aprobatę, gdybym mógł przekonać mojego szefa do twierdzeń zespołu o *kluczowej inwestycji w oprogramowanie i centralizacji wiedzy biznesowej*. Prawdą jest, że moja praca byłaby łatwiejsza, gdybym *mógł coś zrobić, by wzbudzić zaufanie i chęć współpracy pomiędzy moimi zespołami a ekspertami biznesowymi*. W każdym razie słyszę, że właśnie to mogę zrobić”.

  Dobry menedżer!

Kimkolwiek jesteś, powinieneś wziąć pod uwagę ważne ostrzeżenie. Aby odnieść sukces z DDD, *będziesz musiał się czegoś nauczyć*. Będzie dużo tego *czegoś*. Nie powinno to być jednak trudne. Jesteś mądry i musisz uczyć się przez cały czas. Pomimo wszystko każdy z nas musi sprostać następującemu wyzwaniu:

> Osobiście jestem zawsze gotowy, by się uczyć,
> chociaż nie zawsze lubię, gdy ktoś mnie poucza.
> — Sir Winston Churchill

Ta książka ma właśnie taki cel: starałem się, by nauczanie było tak przyjemne, jak to możliwe, i by zapewniało wiedzę niezbędną do skutecznego zastosowania technik DDD.

Czytelnik mógłby jednak postawić pytanie: „Dlaczego należy stosować DDD?". To dobre pytanie.

## Dlaczego należy stosować DDD?

Właściwie już wcześniej podałem kilka dobrych powodów, dla których zastosowanie DDD ma tak bardzo praktyczny sens. Ryzykując złamanie zasady **DRY** (ang. *Don't Repeat Yourself* — dosł. *nie powtarzaj się*), ponownie przytoczę je w tym miejscu oraz dodam do nich kilka nowych powodów. Czy ktoś słyszał echo?

- Umieść ekspertów dziedziny i deweloperów na tym samym poziomie. Dzięki temu wytworzone oprogramowanie będzie miało sens także dla ludzi biznesu, a nie tylko dla koderów. Nie oznacza to jedynie tolerowania przeciwnej grupy. Oznacza stworzenie jednego spójnego i zwięzłego zespołu.
- Fraza „ma sens dla ludzi biznesu" oznacza zainwestowanie w biznes poprzez stworzenie oprogramowania, które maksymalnie będzie przypominać system, jaki stworzyliby biznesowi liderzy i eksperci dziedziny, gdyby sami byli koderami.
- Możemy nauczyć wszystkich członków zespołu więcej o biznesie. Żaden ekspert dziedziny, żaden menedżer szczebla C, nikt i nigdy nie wie wszystkiego na temat biznesu. Poznawanie biznesu jest stałym procesem odkrywania, który z czasem staje się coraz bardziej wnikliwy. Dzięki zastosowaniu DDD wszyscy się uczą, ponieważ każdy coś wnosi do odkrywczych dyskusji.
- Centralizowanie wiedzy ma znaczenie kluczowe, gdyż dzięki temu można zagwarantować, że rozumienie oprogramowanie nie będzie zamkniętą „wiedzą plemienną", dostępną tylko dla kilku wybrańców — zazwyczaj deweloperów.

- Nie ma niepotrzebnych tłumaczeń przekazu pomiędzy ekspertami dziedziny, deweloperami oprogramowania a oprogramowaniem. Nie oznacza to, że istnieje kilka tłumaczeń. Oznacza to dokładnie zero tłumaczeń, ponieważ zespół rozwija wspólny język, którym posługują się wszyscy jego członkowie.
- Projekt jest kodem, a kod jest projektem. Projekt opisuje to, jak system działa. Poznawanie najlepszych projektów kodu następuje poprzez szybkie doświadczalne modele tworzone z wykorzystaniem zwinnych procesów odkrywania.
- Metodologia DDD dostarcza solidnych technik rozwoju oprogramowania, które dotyczą projektu zarówno na poziomie strategicznym, jak i taktycznym. Projekt strategiczny pomaga nam zrozumieć, co stanowi najważniejszą inwestycję w oprogramowanie, jakie istniejące zasoby oprogramowania można wykorzystać, aby zrealizować ją najszybciej i najbezpieczniej, oraz jakie osoby powinny być w nią zaangażowane. Projekt taktyczny pomaga w utworzeniu pojedynczego, eleganckiego modelu rozwiązania z wykorzystaniem przetestowanych bloków budulcowych oprogramowania.

Tak jak każda dobra inwestycja, która ma przynieść duże korzyści, z zastosowaniem metodologii DDD wiążą się pewne początkowe koszty dotyczące czasu i wysiłków zespołu. Biorąc pod uwagę typowe wyzwania, jakie napotykamy w każdym przedsięwzięciu związanym z rozwojem oprogramowania, potrzeba inwestowania w solidną metodologię wytwarzania oprogramowania wydaje się szczególnie ważna.

### *Dostarczanie wartości biznesowych może być ulotne*

Rozwijanie oprogramowania, które dostarcza prawdziwej wartości dla biznesu, nie jest tym samym co rozwijanie zwykłego biznesowego oprogramowania. Oprogramowanie dostarczające prawdziwej wartości biznesowej można porównać ze strategicznymi inicjatywami w biznesie; przynosi ono rozwiązania, w których można wyraźnie zidentyfikować konkurencyjną przewagę — oprogramowanie, którego sednem nie jest technologia, ale biznes.

Wiedza biznesowa nigdy nie jest scentralizowana. Zespoły programistów muszą zachować równowagę pomiędzy potrzebami i żądaniami wielu interesariuszy oraz określić ich priorytety, a ponadto zaangażować wiele osób o różnorodnych umiejętnościach. Celem wszystkich tych przedsięwzięć powinno być odkrywanie funkcjonalnych i niefunkcjonalnych wymagań stawianych oprogramowaniu. Jak po zebraniu wszystkich tych informacji można uzyskać pewność, że określone wymaganie dostarcza prawdziwej wartości biznesowej? Jak się dowiedzieć, jakich wartości biznesowych szukamy, jak je odkrywamy, nadajemy im priorytety i je realizujemy?

Jednym z najbardziej dotkliwych problemów podczas realizowania zadań z zakresu rozwoju oprogramowania biznesowego jest luka pomiędzy ekspertami

dziedziny a deweloperami oprogramowania. Ogólnie rzecz biorąc, prawdziwi eksperci dziedziny są skupieni na dostarczaniu wartości biznesowych. Z drugiej strony deweloperzy oprogramowania są zwykle skoncentrowani na technologii oraz technicznych rozwiązaniach problemów biznesowych. Nie oznacza to, że deweloperzy oprogramowania mają nieodpowiednie motywacje. Chodzi o to, że to technika przyciąga ich uwagę. Nawet kiedy deweloperzy oprogramowania zaczynają współpracować z ekspertami dziedziny, to współpraca przebiega głównie powierzchownie, a w tworzonym oprogramowaniu zazwyczaj stosowane są tłumaczenia (odwzorowania) związane z tym, w jaki sposób myśli ekspert dziedziny, i tym, jak to interpretuje deweloper oprogramowania. Powstałe w ten sposób oprogramowanie zazwyczaj nie odzwierciedla rozpoznawalnej realizacji mentalnego modelu ekspertów dziedziny albo robi to tylko częściowo. Z czasem ten rozdźwięk staje się kosztowny. Tłumaczenie wiedzy dziedzinowej na oprogramowanie gubi się, gdy deweloperzy przechodzą do innych projektów lub odchodzą z firmy.

Inny, choć powiązany z poprzednim, problem powstaje w sytuacji, kiedy eksperci dziedziny nie zgadzają się ze sobą. To się czasami zdarza, ponieważ każdy ekspert ma mniej bądź więcej doświadczenia w określonej modelowanej dziedzinie, a niekiedy są to eksperci z powiązanych ze sobą, ale jednak innych obszarów. Typowy dla wielu ekspertów dziedziny jest również brak wiedzy specjalistycznej z zakresu określonej dziedziny. Czasem się zdarza, że tacy „eksperci" są w większym stopniu analitykami biznesowymi, choć oczekuje się od nich rzeczowego wkładu w dyskusję. Gdy taka sytuacja wymknie się spod kontroli, w rezultacie zamiast ostrych powstają rozmyte modele mentalne — co prowadzi do kolidujących ze sobą modeli oprogramowania.

Jeszcze gorsza sytuacja występuje w przypadku, gdy zastosowanie technicznego podejścia do rozwoju oprogramowania powoduje złą zmianę sposobu, w jaki działa biznes. Chociaż jest to trochę inny scenariusz, to dobrze znane są sytuacje, gdy wdrożenie oprogramowania ERP zmienia ogólny sposób działania biznesu w organizacji, tak by dopasować się do sposobu funkcjonowania oprogramowania ERP. Całkowitego kosztu posiadania ERP nie można w pełni obliczyć w kategoriach opłat licencyjnych i konserwacyjnych. Reorganizacja i zakłócenie działania biznesu mogą być o wiele bardziej kosztowne niż obydwa wymienione czynniki namacalne. Podobna dynamika występuje w sytuacji, kiedy zespół rozwoju oprogramowania interpretuje potrzeby biznesowe i przekształca je na faktyczne działania powstającego oprogramowania. Dla biznesu, klientów firmy i jej partnerów może to powodować powstanie zarówno kosztów, jak i zakłóceń w pracy. Co więcej, ta techniczna interpretacja jest i niepotrzebna, i możliwa do uniknięcia. Wystarczy zastosować sprawdzone techniki wytwarzania oprogramowania. Rozwiązanie to jest kluczową inwestycją.

### *W jaki sposób pomaga DDD?*

DDD jest podejściem do rozwijania oprogramowania, które skupia się na następujących trzech najważniejszych aspektach:

1. DDD łączy ze sobą ekspertów dziedziny i deweloperów oprogramowania, stawiając przed nimi zadanie wytwarzania oprogramowania, które odzwierciedla model mentalny ekspertów biznesowych. Nie oznacza to skoncentrowania wysiłków na modelowaniu „rzeczywistego świata". Zamiast niego DDD dostarcza model, który jest najbardziej przydatny dla biznesu. Czasami przydatne i realistyczne modele przenikają się wzajemnie, ale do momentu, w którym te modele się rozchodzą, DDD stosuje model najbardziej użyteczny.

   Biorąc pod uwagę ten cel, eksperci dziedziny i deweloperzy oprogramowania wspólnie rozwijają Język Wszechobecny tych obszarów biznesu, które są modelowane. Język Wszechobecny jest rozwijany przy akceptacji całego zespołu. Członkowie zespołu porozumiewają się nim i jest on bezpośrednio uwzględniony w modelu oprogramowania. Warto powtórzyć, że zespół składa się zarówno z ekspertów dziedziny, jak i deweloperów oprogramowania. Nigdy nie jest tak, że są „my" i „oni". Zespół to zawsze my. To jest kluczowa wartość biznesowa, która pozwala przetrwać specjalistycznej wiedzy biznesowej dłużej, niż trwają stosunkowo krótkie początkowe prace rozwojowe związane z kilkoma pierwszymi wersjami oprogramowania. Wiedza ta trwa także dłużej, niż istnieją zespoły, które to oprogramowanie wytworzyły. To jest element, który udowadnia, że koszt rozwoju oprogramowania jest uzasadnioną inwestycją biznesową, a nie tylko zwykłym kosztem.

   Podejmowany wysiłek jednoczy ekspertów dziedziny, którzy początkowo nie zgadzają się ze sobą albo nie mają podstawowej wiedzy z zakresu dziedziny. Co więcej, praca nad oprogramowaniem wzmacnia spójność zespołu dzięki rozpowszechnianiu obszernej wiedzy dziedzinowej wśród wszystkich członków zespołu, w tym deweloperów oprogramowania. Wysiłki projektowe można uznać za szkolenie mówiące o tym, że każda firma powinna inwestować w swoich pracowników sektora wiedzy.

2. DDD zajmuje się strategicznymi inicjatywami biznesu. Chociaż to strategiczne podejście do projektowania w naturalny sposób zawiera analizę techniczną, to w większym stopniu skupia się na strategicznym kierunku biznesu. Pomaga to zdefiniować najlepsze wewnątrzzespołowe relacje organizacyjne i dostarcza system wczesnego ostrzegania, pozwalający rozpoznać sytuacje, w których określona relacja mogłaby spowodować niepowodzenie oprogramowania, a nawet całego projektu. Celem technicznych aspektów projektowania strategicznego jest czytelne wytyczenie granic systemów i obszarów biznesowych, co pozwala chronić wszystkie *usługi na poziomie biznesu*. To dostarcza istotnych motywów do osiągnięcia ogólnej *architektury ukierunkowanej na usługi* lub architektury sterowanej względami biznesowymi.

3. DDD wychodzi naprzeciw rzeczywistym technicznym wymaganiom oprogramowania poprzez wykorzystanie narzędzi projektowania taktycznego do analizy i rozwoju wykonywalnych produktów oprogramowania. Narzędzia projektowania taktycznego pozwalają deweloperom produkować oprogramowanie, które jest poprawną kodyfikacją mentalnego modelu ekspertów dziedziny. Jest ono wysoce testowalne, mniej podatne na błędy (tworzy twierdzenie, które da się udowodnić), zgodne z umowami dotyczącymi zapewnienia jakości usług (SLA) i skalowalne, a także umożliwia wykonywanie rozproszonych obliczeń. Ogólnie rzecz biorąc, najlepsze praktyki DDD dotyczą kilkunastu wysokopoziomowych aspektów architektury oraz niższych poziomów projektu. Główny akcent kładą one na rozpoznawanie rzeczywistych reguł biznesowych i niezmienników danych oraz na ochronę przed występowaniem błędów.

   Stosowanie tego podejścia w rozwoju oprogramowania pozwala wszystkim członkom zespołu skutecznie dostarczać rzeczywistą wartość biznesową.

### *„Mocowanie się" ze złożonością dziedziny*

Metod DDD chcemy użyć głównie w tych obszarach, które są najważniejsze dla biznesu. Nie inwestujemy w to, co może być łatwo zastąpione. *Inwestujemy w nietrywialne, bardziej złożone komponenty. Najbardziej wartościowe i najważniejsze rzeczy, które dają największe szanse na osiągnięcie zysków.* Właśnie dlatego nazywamy taki model **Dziedziną Główną (2)**. To właśnie ona oraz, w dalszej kolejności, ważniejsze **Poddziedziny Pomocnicze (2)** zasługują na największe inwestycje i je otrzymują. W związku z tym bez wątpienia należałoby się dowiedzieć, co to znaczy „złożony".

> **Używaj DDD, aby upraszczać, a nie komplikować**
>
> Metod DDD należy używać do modelowania złożonych dziedzin w najprostszy możliwy sposób. Nigdy nie należy stosować DDD, by bardziej skomplikować istniejące rozwiązanie.

To, co można określić jako „złożone", będzie różne dla różnych rodzajów biznesu. Różne firmy mają różne wyzwania, różne poziomy dojrzałości i różne zdolności rozwijania oprogramowania. Dlatego zamiast ustalania tego, co jest złożone, łatwiejsze może być określenie tego, co jest *nietrywialne*. A zatem *Twój zespół i kierownictwo powinni ustalić, czy system, nad którym planujecie pracować, zasługuje na poniesienie kosztów związanych z inwestycją w DDD.*

**Karty punktów DDD.** Aby ustalić, czy określony projekt kwalifikuje się do zainwestowania w DDD, można skorzystać z tabeli 1.1. Jeżeli opis w wierszu na karcie punktów pasuje do Twojego projektu, umieść odpowiednią liczbę punktów w kolumnie po prawej stronie. Policz wszystkie punkty dla swojego projektu. Jeśli jest ich 7 albo więcej, poważnie się zastanów nad korzystaniem z DDD.

**Tabela 1.1.** Karty punktów DDD

*Czy Twój projekt uzyskał 7 lub więcej punktów?*

| Jeżeli Twój projekt... | Punkty | Uwagi pomocnicze | Twój wynik |
|---|---|---|---|
| Jeżeli Twoja aplikacja jest w całości skoncentrowana na danych i kwalifikuje się do zastosowania czystego rozwiązania CRUD, gdzie każda operacja jest zasadniczo prostym zapytaniem bazy danych w celu stworzenia, odczytania, zaktualizowania albo usunięcia rekordu, to nie potrzebujesz DDD. Twój zespół powinien jedynie opracować wygodny interfejs do edytora tabel bazy danych. Mówiąc inaczej, jeżeli możesz zaufać swoim użytkownikom, że będą prawidłowo wprowadzali dane bezpośrednio do tabeli, aktualizowali je i czasami usuwali, to nie potrzebujesz nawet interfejsu użytkownika. Taki scenariusz nie jest realistyczny, ale można go sobie wyobrazić. Jeśli do stworzenia rozwiązania można użyć prostego narzędzia do programowania aplikacji bazodanowych, to nie warto marnować czasu i pieniędzy firmy na zastosowanie DDD. | 0 | Choć wydaje się to oczywiste, odróżnienie czegoś, co jest proste, od złożonego nie zawsze jest łatwe. To nie jest tak, że każda aplikacja, które nie jest czystym rozwiązaniem CRUD, zasługuje na czas i wysiłek włożone w stosowanie DDD. W związku z tym niekiedy trzeba użyć innych metryk, które pozwolą narysować linię pomiędzy tym, co jest złożone, a tym, co złożone nie jest. | |
| Jeżeli Twój system wymaga wykonania tylko 30 (albo mniej) operacji biznesowych, to jest prawdopodobnie dość prosty. Oznaczałoby to, że w Twojej aplikacji byłoby łącznie nie więcej niż 30 historyjek użytkowników lub że aplikacja wykorzystuje przepływy spraw, a każdy z tych przepływów obejmuje tylko minimalny zakres logiki biznesowej. Jeśli można szybko i łatwo stworzyć taką aplikację, używając Ruby on Rails albo Groovy i Grails, i jeśli brak kontroli nad złożonością i zmianami nie nastręcza Ci problemów, to Twój system prawdopodobnie nie potrzebuje zastosowania DDD. | 1 | Dla jasności: mam na myśli około 25 – 30 pojedynczych metod biznesowych, a nie 25 – 30 kompletnych interfejsów usługowych, z których każdy obejmuje wiele metod. Ten drugi system może być złożony. | |

| | | |
|---|---|---|
| Można zatem powiedzieć, że jeśli system zawiera 30 – 40 historyjek użytkownika albo przepływów spraw, to zaczyna być złożony. W takim przypadku Twój system zaczyna zmierzać w kierunku obszaru zajmowanego przez DDD. | 2 | *Ostrzeżenie*: bardzo często złożoność nie jest rozpoznawana wystarczająco wcześnie. *My, deweloperzy oprogramowania, jesteśmy naprawdę dobrzy w niedocenianiu złożoności i poziomu wysiłku*. Samo to, że mamy ochotę stworzyć aplikację Rails albo Grails, wcale nie znaczy, że powinniśmy. W dłuższej perspektywie może to bardziej zaszkodzić niż pomóc. |
| Nawet jeśli aplikacja nie jest złożona teraz, to czy jej złożoność może wzrosnąć? Możemy nie wiedzieć tego na pewno do czasu, aż prawdziwi użytkownicy aplikacji zaczną jej używać. W kolumnie „Uwagi pomocnicze" zamieszczono opis czynności, które mogą pomóc w odkryciu prawdziwej sytuacji.<br>Należy tu zachować ostrożność. Jeżeli istnieją jakiekolwiek przesłanki co do tego, że aplikacja ma nawet umiarkowaną złożoność — to jest dobre miejsce, by być trochę paranoikiem — może to być wystarczający znak tego, że będzie ona bardziej niż umiarkowanie złożona. W tym przypadku powinniśmy się skłonić ku DDD. | 3 | W tym przypadku opłaca się przeanalizować z ekspertami dziedziny bardziej złożone scenariusze użycia i zobaczyć, dokąd to zaprowadzi. Czy eksperci dziedziny:<br>już proszą o bardziej złożone cechy funkcjonalne? W takim razie istnieje prawdopodobieństwo, że aplikacja już teraz albo wkrótce stanie się zbyt złożona, by można było używać podejścia CRUD;<br>są tak znudzeni cechami funkcjonalnymi, że ledwie znoszą konieczność ich omawiania? Prawdopodobnie aplikacja nie jest złożona. |
| Cechy funkcjonalne aplikacji przez lata będą się często zmieniać. Trudno przewidzieć, czy te zmiany będą proste. | 4 | Zastosowanie DDD może pomóc w zarządzaniu złożonością refaktoryzacji modelu w miarę upływu czasu. |
| Nie rozumiesz **Dziedziny (2)**, ponieważ jest nowa. Z tego, co wiecie Ty i członkowie Twojego zespołu, nikt tego wcześniej nie zrobił. Najprawdopodobniej oznacza to, że aplikacja jest złożona albo przynajmniej zasługuje na uważną analizę zmierzającą do określenia poziomu złożoności. | 5 | Trzeba popracować z ekspertami dziedziny i poeksperymentować z modelami, by opracować ten właściwy. Na pewno też przypisałeś punkty w jednym albo kilku poprzednich kryteriach, więc zastosuj DDD. |

To ćwiczenie związane z punktacją mogło doprowadzić Twój zespół do następujących wniosków:

To źle, że nie możemy szybko i łatwo zmienić biegu, gdy odkryjemy, że znajdujemy się po niewłaściwej stronie złożoności, niezależnie od tego, czy niewłaściwa strona jest mniej, czy bardziej złożona, niż sądziliśmy.

To oczywiste, ale oznacza tylko to, że musimy osiągnąć lepszą sprawność w określaniu prostoty i złożoności we wczesnej fazie planowania projektu. To pozwoli zaoszczędzić nam dużo czasu, kosztów i kłopotów.

Kiedy podejmiemy ważną decyzję architektoniczną i mocno zaangażujemy się w rozwój kilku przypadków użycia, z pewnością będziemy mocno zablokowani zastosowaną architekturą. Lepiej dokonywać mądrych wyborów odpowiednio wcześnie.

Jeżeli któraś z tych obserwacji pasuje do Twojego zespołu, oznacza to, że robisz właściwy użytek z krytycznego myślenia.

### *Anemia i utrata pamięci*

Anemia może być poważną dolegliwością zdrowotną z niebezpiecznymi skutkami ubocznymi. Gdy po raz pierwszy w literaturze pojawił się termin *Anemiczny Model Dziedziny* (ang. *Anemic Domain Model*) [Fowler, Anemic] *nie miał on oznaczać pozytywnej cechy*. To tak, jakby powiedzieć, że model dziedziny, który jest słaby, bez wrodzonych behawioralnych mocnych stron, mógłby być czymś dobrym. O dziwo Anemiczne Modele Dziedziny pojawiły się w naszej branży. Kłopot w tym, że większość deweloperów uważa je za całkiem normalne i nawet nie przyznaje, że w systemach korzystających z takich modeli istnieje poważny problem. Ale to jest prawdziwy problem.

Zastanawiasz się nad tym, czy Twój model jest „zmęczony", „apatyczny", „zapomina", jest „niezdarny" i „potrzebuje klepnięcia w ramię"? Jeżeli nagle doświadczasz technicznej hipochondrii, to nadszedł właściwy moment na przeprowadzenie samobadania. Albo się uspokoisz, albo potwierdzisz swoje najgorsze obawy. W celu przeprowadzenia badania wykonaj czynności zapisane w tabeli 1.2.

Jakich odpowiedzi udzieliłeś?

Jeżeli odpowiedziałeś „nie" na oba pytania, Twój model dziedziny ma się całkiem dobrze.

Jeżeli odpowiedziałeś „tak" na oba pytania, Twój model dziedziny jest bardzo, bardzo chory. To jest anemia. Dobra wiadomość jest taka, że możesz mu pomóc, jeśli będziesz kontynuować lekturę.

Jeżeli odpowiedziałeś „tak" na jedno pytanie i „nie" na drugie, to albo zaprzeczasz sam sobie, albo cierpisz na złudzenia lub inne neurologiczne dolegliwości, których przyczyną może być anemia. Co powinieneś zrobić, jeśli odpo-

**Tabela 1.2.** Określenie historii zdrowia modelu dziedziny

| | Tak/nie |
|---|---|
| Czy w Twoim oprogramowaniu, które nazywasz modelem dziedziny, w większości występują publiczne gettery i settery i nie ma lub prawie nie ma żadnej logiki biznesowej — tzn. występują obiekty, które w większości są kontenerami wartości atrybutów? | |
| Czy komponenty oprogramowania, które często używają Twojego modelu dziedziny, są tymi, w których występuje większość logiki biznesowej Twojego systemu i które wykorzystują często publiczne gettery i settery modelu dziedziny? Prawdopodobnie nazywasz tę konkretną warstwę klienta modelu dziedziny **Warstwą Usług** albo **Warstwą Aplikacji (4, 14)**. Jeżeli zamiast tego opisuje to Twój interfejs użytkownika, odpowiedz „tak" na to pytanie i napisz tysiąc razy na tablicy, że nigdy więcej tego nie zrobisz. | |
| **Wskazówka: właściwe odpowiedzi to albo „tak" na oba pytania, albo „nie" na oba pytania.** | |

wiedzi kolidują ze sobą? Od razu przejdź do pierwszego pytania i ponownie uruchom samosprawdzenie. Nie spiesz się, ale pamiętaj, że odpowiedzią na oba pytania powinno być dobitne „tak!".

Zgodnie z tym, co powiedział Fowler [Fowler, Anemic], Anemiczny Model Dziedziny jest zły, ponieważ wymaga poniesienia wysokich kosztów rozwijania modelu dziedziny, a w zamian uzyskujemy niewiele korzyści albo wręcz nie uzyskujemy żadnych. Na przykład ze względu na niedopasowanie obiektowo-relacyjne deweloperzy takiego modelu dziedziny poświęcają zbyt dużo czasu i wysiłku na mapowanie obiektów do trwałego magazynu i w drugą stronę. To wysoka cena, jaką trzeba zapłacić, podczas gdy w zamian otrzymujemy niewiele korzyści lub nie uzyskujemy ich wcale. Dodam, że to, czym dysponujemy, w ogóle nie jest modelem dziedziny, a tylko modelem danych zrzutowanym z modelu relacyjnego (albo innego modelu bazodanowego) na obiekty. To jest oszukany model dziedziny, któremu właściwie bliżej do definicji Aktywnego Rekordu [Fowler, P of EAA]. Najprawdopodobniej taką architekturę można uprościć. Wystarczy zrezygnować z pretensjonalności i przyznać, że w istocie używamy pewnej formy Skryptu Transakcji [Fowler, P of EAA].

## Przyczyny anemii

Jeżeli więc Anemiczny Model Dziedziny jest chorym rezultatem straconego wysiłku projektowego, to dlaczego tak wielu deweloperów go stosuje, myśląc, że ich model znajduje się w doskonałym zdrowiu? Na pewno to odzwierciedla mentalność programowania proceduralnego, ale nie sądzę, aby to był najważniejszy powód. Wielu deweloperów z naszej branży to naśladowcy przykładów kodu, co w gruncie rzeczy nie jest złe, o ile przykłady są wysokiej jakości. Często jednak

przykłady kodu są skoncentrowane celowo na zademonstrowaniu określonego pojęcia albo funkcji interfejsu programowania aplikacji (API) w najprostszy możliwy sposób, bez zwracania uwagi na zasady dobrego projektowania. Pomimo to nadmiernie uproszczony kod przykładów, w których zwykle występuje mnóstwo getterów i setterów, jest kopiowany każdego dnia bez zwracania zbytniej uwagi na projekt.

Istnieje jeszcze jeden, o wiele starszy wpływ. „Starożytna" historia języka Visual Basic firmy Microsoft ma dużo wspólnego z tym, z czym mamy dziś do czynienia. Nie twierdzę, że Visual Basic był złym językiem i zintegrowanym środowiskiem programisty (IDE), ponieważ zawsze było to wysoce produktywne środowisko i w pewnym sensie wpłynęło na rozwój branży. Oczywiście niektórzy deweloperzy być może unikali jego bezpośredniego oddziaływania, ale ostatecznie Visual Basic pośrednio wywarł wpływ na prawie każdego dewelopera oprogramowania. Zwróćmy uwagę na przebieg czasowy pokazany w tabeli 1.3.

**Tabela 1.3.** Przebieg czasowy od Bogatych Zachowań do Niesławnej Anemii

| **Lata osiemdziesiąte** | **1991** | **1992 – 1995** | **1996** | **1997** | **1998 –** |
|---|---|---|---|---|---|
| Wpływ programowania obiektowego dzięki rozwojowi języków Smalltalk i C ++ | Właściwości i arkusze właściwości języka Visual Basic | Wizualne narzędzia i środowiska IDE stają się powszechne | Opublikowano oprogramowanie Java JDK 1.0 | Specyfikacja JavaBean | Eksplozja narzędzi bazujących na refleksjach korzystających z właściwości dla platform Java i .NET |

Mówię o wpływie, jaki wywarły na branżę właściwości i arkusze właściwości, dostępne zarówno za pośrednictwem getterów i setterów, jak i za pośrednictwem projektanta formularzy Visual Basica — narzędzia, dzięki któremu zdobyły tak wielką popularność. Wystarczyło umieścić kilka niestandardowych egzemplarzy kontrolek na formularzu, wypełnić ich arkusze właściwości i voilà! Powstawała w pełni działająca aplikacja Windows. To zajmowało zaledwie kilka minut, a nie kilka dni, jakie były potrzebne do zaprogramowania podobnej aplikacji przy bezpośrednim wykorzystaniu API systemu Windows i języka C.

Ale co to wszystko ma wspólnego z Anemicznymi Modelami Dziedziny? *Standard Java-Bean początkowo opracowano po to, by pomóc w tworzeniu wizualnych narzędzi programowania dla Javy*. Motywacją do jego powstania było przeniesienie możliwości technologii Microsoft ActiveX na platformę Javy. To stworzyło nadzieję na stworzenie rynku pełnego różnego rodzaju kontrolek — podobnych do tych tworzonych w Visual Basicu — produkowanych przez różnych dostawców. Wkrótce do „wagonu" JavaBean wskoczyły prawie wszystkie

frameworki i biblioteki. W tej grupie znalazły się większa część pakietu Java SDK/JDK, jak również takie biblioteki jak popularna biblioteka Hibernate. Z punktu widzenia zagadnień związanych z DDD bibliotekę *Hibernate opracowano jako narzędzie utrwalania modeli dziedziny*. Trend utrzymywał się do czasu pojawienia się platformy .NET.

Co ciekawe, dowolny model dziedziny, który został utrwalony z wykorzystaniem biblioteki Hibernate, w początkowym okresie jej istnienia musiał udostępniać publiczne gettery i settery zarówno dla każdego prostego atrybutu, jak i dla złożonej asocjacji w każdym obiekcie dziedziny. To oznaczało, że nawet jeśli ktoś chciał zaprojektować obiekt **POJO** (ang. *Plain Old Java Object* — dosł. *Prosty stary obiekt Javy*) z interfejsem obfitującym w zachowania, to musiał wewnętrzne atrybuty udostępniać publicznie, tak by biblioteka Hibernate mogła utrwalić, a później odtworzyć obiekty dziedziny. Oczywiście były możliwości ukrycia publicznego interfejsu JavaBean, ale ogólnie rzecz biorąc, deweloperzy w większości nie zastanawiali się, dlaczego mieliby to robić (ani nawet tego nie rozumieli).

> **Czy korzystając z DDD, powinienem zastanawiać się nad używaniem maperów obiektowo-relacyjnych?**
>
> Wcześniejsza krytyka biblioteki Hibernate wynika z perspektywy historycznej. Już od dość długiego czasu biblioteka Hibernate wspiera wykorzystanie ukrytych getterów i setterów, a nawet bezpośredni dostęp do pól. W kolejnych rozdziałach pokażę, jak uniknąć anemii w tworzonych modelach w przypadku stosowania biblioteki Hibernate i innych mechanizmów utrwalania. Spokojna głowa!

Większość, jeżeli nie wszystkie, frameworków webowych działa wyłącznie na bazie standardu JavaBean. Jeżeli obiekty Javy mają mieć możliwość wypełniania stron WWW, korzystniej by było, aby zapewniały lepsze wsparcie dla specyfikacji JavaBean. Jeśli formularze HTML mają wypełniać obiekty Javy po przesłaniu na stronę serwera, korzystniej by było, żeby obiekty formularzy Javy zapewniały lepsze wsparcie dla specyfikacji JavaBean.

Niemal każdy framework dostępny dziś na rynku wymaga używania publicznych właściwości prostych obiektów, promując je tym samym. Większość deweloperów nie ma innego wyjścia. Musi ulec wpływom wszystkich „anemicznych" klas wykorzystywanych w ich przedsiębiorstwach. Trzeba się do tego przyznać. Czy nie dotyczy to także Ciebie, Czytelniku? W rezultacie mamy sytuację, którą najlepiej opisuje etykieta *anemia wszędzie*.

## Jakie szkody wyrządza anemia Twojemu modelowi?

Załóżmy więc, że zgodziliśmy się z tym, iż anemia jest faktem i jest dla nas irytująca. Co *anemia wszędzie* ma wspólnego z *utratą pamięci*? Co zazwyczaj widzimy, gdy czytamy kod klienta Anemicznego Modelu Dziedziny (na przykład oszukańczej *Usługi Aplikacji* [4, 14] à la Skrypt Transakcji)? Oto prosty przykład:

```
@Transactional
public void saveCustomer(
    String customerId,
    String customerFirstName, String customerLastName,
    String streetAddress1, String streetAddress2,
    String city, String stateOrProvince,
    String postalCode, String country,
    String homePhone, String mobilePhone,
    String primaryEmailAddress, String secondaryEmailAddress) {

    Customer customer = customerDao.readCustomer(customerId);

    if (customer == null) {
        customer = new Customer();
        customer.setCustomerId(customerId);
    }

    customer.setCustomerFirstName(customerFirstName);
    customer.setCustomerLastName(customerLastName);
    customer.setStreetAddress1(streetAddress1);
    customer.setStreetAddress2(streetAddress2);
    customer.setCity(city);
    customer.setStateOrProvince(stateOrProvince);
    customer.setPostalCode(postalCode);
    customer.setCountry(country);
    customer.setHomePhone(homePhone);
    customer.setMobilePhone(mobilePhone);
    customer.setPrimaryEmailAddress(primaryEmailAddress);
    customer.setSecondaryEmailAddress (secondaryEmailAddress);

    customerDao.saveCustomer(customer);
}
```

**Celowo prosty przykład**

Co prawda ten przykład nie pochodzi z bardzo interesującej dziedziny, ale pomaga zbadać daleki od ideału projekt i zdecydować, jak go zrefaktoryzować, tak by stał się znacznie lepszy. Powiedzmy jasno, że to ćwiczenie nie ma na celu doprowadzić nas do lepszego sposobu zapisywania danych. Chodzi o tworzenie modelu oprogramowania, które dodaje wartość do Twojego biznesu, mimo że ten przykład może nie wydawać się wartościowy.

Co ten kod właściwie robi? W istocie jest on dość uniwersalny. Zapisuje obiekt `Customer` niezależnie od tego, czy jest on nowy, czy istniał wcześniej. Zapisuje obiekt `Customer` niezależnie od tego, czy zmieniło się nazwisko, czy też osoba przeprowadziła się do nowego domu. Zapisuje obiekt `Customer` bez względu na to, czy osoba otrzymała nowy numer telefonu domowego, czy zrezygnowała z usług telekomunikacyjnych, oraz niezależnie od tego, czy po raz pierwszy otrzymała telefon komórkowy, czy miały miejsce obie te sytuacje naraz. Zapisuje obiekt `Customer` nawet wtedy, gdy osoba zaczęła używać Gmaila zamiast Juno albo zmieniła pracę i teraz korzysta z służbowego adresu e-mail. Wow! Co za niezwykła metoda!

Czy rzeczywiście? Właściwie nie mamy żadnych informacji na temat tego, w jakich sytuacjach biznesowych ta metoda `saveCustomer()` jest używana — przynajmniej nie do końca. Dlaczego ta metoda została w ogóle utworzona? Czy ktokolwiek pamięta jej pierwotne przeznaczenie oraz wszystkie motywacje do jej zmieniania, by udzielić wsparcia dla szerokiego wachlarza celów biznesowych? Te wspomnienia z pewnością zostały utracone zaledwie kilka tygodni albo miesięcy po tym, jak metodę stworzono, a potem zmodyfikowano. Jest jeszcze gorzej. Trudno w to uwierzyć? Przyjrzyjmy się kolejnej wersji tej samej metody:

```
@Transactional
public void saveCustomer(
    String customerId,
    String customerFirstName, String customerLastName,
    String streetAddress1, String streetAddress2,
    String city, String stateOrProvince,
    String postalCode, String country,
    String homePhone, String mobilePhone,
    String primaryEmailAddress, String secondaryEmailAddress) {

    Customer customer = customerDao.readCustomer(customerId);

    if (customer == null) {
        customer = new Customer();
        customer.setCustomerId(customerId);
    }
    if (customerFirstName != null) {
        customer.setCustomerFirstName(customerFirstName);
    }
    if (customerLastName != null) {
        customer.setCustomerLastName(customerLastName);
    }
    if (streetAddress1 != null) {
        customer.setStreetAddress1(streetAddress1);
    }
    if (streetAddress1 != null) {
        customer.setStreetAddress2(streetAddress2);
    }
    if (city != null) {
        customer.setCity(city);
    }
    if (stateOrProvince != null) {
        customer.setStateOrProvince(stateOrProvince);
    }
    if (postalCode != null) {
        customer.setPostalCode(postalCode);
    }
    if (country != null) {
        customer.setCountry(country);
    }
    if (homePhone != null) {
        customer.setHomePhone(homePhone);
    }
    if (mobilePhone != null) {
        customer.setMobilePhone(mobilePhone);
```

```
    }
    if (primaryEmailAddress != null) {
        customer.setPrimaryEmailAddress(primaryEmailAddress);
    }
    if (secondaryEmailAddress != null) {
        customer.setSecondaryEmailAddress (secondaryEmailAddress);
    }

    customerDao.saveCustomer(customer);
}
```

Muszę tutaj zaznaczyć, że ten przykład nie jest taki zły, jak się wydaje. Często się zdarza, że kod mapowania danych ma dość złożoną formę i zawiera wewnątrz sporo logiki biznesowej. Oszczędziłem Ci, Czytelniku, w tym przykładzie najgorszego, ale prawdopodobnie dostrzegłeś to sam.

Teraz każdy z parametrów innych niż `customerId` jest opcjonalny. Możemy więc użyć tej metody, by zapisać obiekt `Customer` w co najmniej kilkunastu sytuacjach biznesowych! Ale czy to naprawdę jest dobre? W jaki sposób moglibyśmy przetestować tę metodę, by zyskać pewność, że w nieodpowiednich sytuacjach obiekt `Customer` nie zostanie zapisany?

Bez wchodzenia w zbyt wiele szczegółów można powiedzieć, że istnieje znacznie więcej sposobów nieprawidłowego działania tej metody niż sposobów jej poprawnego działania. Być może istnieją ograniczenia bazy danych, które zapobiegają utrwalaniu zupełnie nieprawidłowego stanu, ale w przypadku takiej postaci, w jakiej kod występuje teraz, musielibyśmy zajrzeć do bazy danych, żeby mieć pewność. Prawie na pewno zajmie nam jakiś czas znalezienie mentalnego odwzorowania pomiędzy atrybutami Javy a nazwami kolumn. Kiedy już sprawdzimy tę część, możemy się dowiedzieć, że ograniczeń bazy danych nie ma albo że są one niekompletne.

Moglibyśmy przyjrzeć się wielu klientom (nie licząc tych, które zostały dodane po zakończeniu tworzenia interfejsu użytkownika w celu zarządzania automatycznymi zdalnymi klientami) i porównać wersje kodu źródłowego, aby uzyskać jakiś pogląd na temat tego, dlaczego kod został zaimplementowany właśnie tak jak teraz. Podczas poszukiwania odpowiedzi dowiemy się, że nikt nie potrafi wyjaśnić, z jakiej przyczyny ta jedna metoda działa tak, jak działa; nikt nie potrafi też powiedzieć, ile istnieje prawidłowych przypadków użycia. Zrozumienie tego bez niczyjej pomocy mogłoby zająć nawet kilka godzin albo dni.

## Logika kowboja

AJ: Ten facet jest tak zdezorientowany, że nie wie, czy nosi na plecach kartofle, czy też jeździ na rolkach w stadzie bawołów.

Eksperci dziedziny nie mogą tutaj pomóc, ponieważ by zrozumieć kod, musieliby być programistami. Nawet jeśli kilku ekspertów dziedziny wie o programowaniu chociaż tyle, aby przynajmniej potrafić przeczytać kod, to prawdopodobnie próbując wyjaśnić, co ten kod ma wspierać, będą tak samo zagubieni jak deweloper. Biorąc pod uwagę te wszystkie względy, zastanówmy się, czy ośmielimy się zmienić ten kod, a jeżeli tak, to jak się do tego zabierzemy.

Istnieją tutaj przynajmniej trzy poważne problemy:

1. Interfejs metody `saveCustomer()` niezbyt dokładnie opisuje cel jej działania.
2. Implementacja metody `saveCustomer()` wprowadza ukrytą złożoność.
3. „Obiekt dziedziny" `Customer` w rzeczywistości wcale nie jest obiektem. Jest tylko nieinteligentnym kontenerem na dane.

Nazwijmy tę sytuację, która jest nie do pozazdroszczenia, *utratą pamięci wywołaną anemią*. Taka sytuacja zdarza się bardzo często podczas pracy nad projektami, w których tworzony jest ten rodzaj niejawnego i całkowicie subiektywnego „projektu" kodu.

> **Wstrzymaj się przez chwilę!**
>
> W tym momencie niektórzy czytelnicy mogliby pomyśleć: „Nasze projekty nigdy nie wykraczają poza białą tablicę. Rysujemy jakąś strukturę, a kiedy osiągniemy porozumienie, możemy swobodnie przejść do implementacji. To przerażające".
>
> Jeśli tak pomyślałeś, spróbuj nie odróżniać projektu od implementacji. Pamiętaj, że jeżeli stosujesz DDD, to *projekt jest kodem, a kod jest projektem*. Mówiąc inaczej, schematy rysowane na białej tablicy nie są projektem, a jedynie sposobem dyskutowania o wyzwaniach modelu.
>
> Czytaj uważnie. W dalszej części tej książki nauczysz się czerpać pomysły z rysunków na białej tablicy i wykorzystywać je w pracy.

W tym momencie czytelnik powinien umieć dostrzegać niebezpieczeństwa związane z używaniem złego kodu i czuć potrzebę tworzenia lepszego projektu. Dobra wiadomość jest taka, że możesz odnieść sukces w opracowywaniu jawnego, dokładnego projektu kodu.

## W jaki sposób stosować DDD?

Spróbujmy odejść na chwilę od trudnych dyskusji na temat implementacji i opowiedzieć o jednej z najpotężniejszych własności metod DDD — Języku Wszechobecnym. Jest to jeden z dwóch najważniejszych filarów DDD. Drugim jest **Kontekst Ograniczony (2)**. Jeden nie może istnieć bez drugiego.

**Pojęcia w kontekście**

Na tę chwilę pomyśl o Kontekście Ograniczonym jak o pojęciowej granicy wokół całej aplikacji albo jak o skończonym systemie. Celem istnienia tej granicy jest podkreślenie tego, że każde użycie określonego pojęcia, frazy lub zdania Wszechobecnego Języka wewnątrz tej granicy ma określone znaczenie w kontekście. Dowolne użycie pojęcia na zewnątrz tej granicy mogłoby mieć, i prawdopodobnie miałoby, całkowicie inne znaczenie. Konteksty Ograniczone zostały szczegółowo opisane w rozdziale 2.

## *Język Wszechobecny*

Język Wszechobecny jest wspólnym językiem, którym posługuje się zespół. Posługują się nim zarówno eksperci dziedziny, jak i deweloperzy. W istocie posługują się nim wszyscy członkowie zespołu projektowego. Niezależnie od tego, jaką funkcję pełnisz w zespole, ze względu na to, że w nim jesteś, posługujesz się Językiem Wszechobecnym projektu.

**Zatem uważasz już, że wiesz, czym jest Język Wszechobecny?**

*Oczywiście, jest to język biznesu.*

Otóż nie.

*Z pewnością musi to być przyjęta terminologia branżowa.*

Nie sądzę.

*Wyraźnie wygląda to na żargon używany przez ekspertów dziedziny.*

Przepraszam, ale nie.

*Język Wszechobecny jest współdzielonym językiem rozwijanym przez zespół — grupę tworzoną zarówno przez ekspertów dziedziny, jak i deweloperów oprogramowania.*

To jest to! Teraz uchwyciliśmy sedno!

Naturalnie eksperci dziedziny mają duży wpływ na Język, ponieważ najlepiej znają tę część biznesu. Mają na niego wpływ także standardy obowiązujące w branży. Jednakże Język *jest w większym stopniu skoncentrowany na tym, według jakich reguł działa sam biznes*. Często również zdarza się, że dwóch lub więcej ekspertów dziedziny nie zgadza się w kwestii pojęć i terminów, z którymi pracują. Są one zazwyczaj źle rozumiane ze względu na to, że wcześniej eksperci niewystarczająco dokładnie przemyśleli każdy z przypadków. Zatem podczas gdy eksperci dziedziny i deweloperzy pracują razem w celu opracowania modelu dziedziny, prowadzą dyskusje zmierzające zarówno do kompromisu, jak i konsensusu, by uzyskać *najlepszy Język dla projektu*, zespół nigdy nie idzie na kompromis w zakresie jakości Języka. Przedmiotem kompromisu są jedynie najlepsze pojęcia, terminy i znaczenia. Początkowa zgoda nie oznacza jednak zgody ostatecznej. Język rozwija się i zmienia w czasie. Osiągane są małe i duże przełomy — zupełnie tak, jak w żywym języku.

Nie chodzi głównie o to, aby deweloperzy „byli po tej samej stronie" co eksperci dziedziny. Język nie jest wyłącznie podzbiorem biznesowego żargonu, którego deweloperzy mają obowiązek używać. To jest prawdziwy Język, tworzony

przez cały zespół: ekspertów dziedziny, deweloperów, analityków biznesowych — wszystkich biorących udział w budowaniu systemu. Punktem wyjścia dla Języka mogą być terminy, które są naturalnym żargonem ekspertów dziedziny. Język nie ogranicza się jednak wyłącznie do nich, ponieważ sam Język z czasem musi się rozwijać. Wystarczy powiedzieć, że kiedy eksperci dziedziny są zaangażowani w tworzenie Języka, często nie zgadzają się do pewnego stopnia w kwestii terminów i znaczeń, o których sądzono, że są już powszechnie przyjęte.

W tabeli 1.4 zaprezentowano model w kodzie „stosowania szczepionek grypy". Kod jest zapisany w Języku, którym zespół powinien się otwarcie posługiwać. Gdy zespół omawia ten aspekt modelu, dosłownie są wypowiadane takie frazy jak „Pielęgniarki podają pacjentom szczepionki grypy w standardowych dawkach".

Tabela 1.4. Analiza najlepszego modelu dla biznesu

***Które określenie jest lepsze dla biznesu?***

***Chociaż zdania drugie i trzecie są podobne, to zastanówmy się, jak powinien być zaprojektowany kod.***

| **Możliwe punkty widzenia** | **Wynikowy kod** |
|---|---|
| *„Kogo to obchodzi? Po prostu zakodujmy to jakoś".*<br>Hm, podejście dalekie od prawidłowego. | `pacjent.ustawTypSzczepionki(TypySzczepionek.TYP_GRYPA);`<br>`pacjent.ustawDawke(dawka);`<br>`pacjent.ustawPielegniarke(pielegniarka);` |
| *„Podajemy pacjentom szczepionki grypy".*<br>Lepiej, ale nie uwzględniamy ważnych pojęć. | `pacjent.podajSzczepionkeGrypy();` |
| *„Pielęgniarka podaje pacjentom szczepionki grypy w standardowych dawkach".*<br>Wydaje się, że to najlepszy sposób — przynajmniej dopóki nie dowiemy się więcej. | `Szczepionka szczepionka = szczepionki.grypaStandardowaDawkaDorosli();`<br>`pielegniarka.podajeSzczepionkeGrypy(pacjent, szczepionka);` |

Podczas ewolucji Języka będą uwidaczniały się drobne rozbieżności pomiędzy rozumieniem pojęć w formie, w jakiej istnieją one w umysłach ekspertów, a tym, co z tego ewoluuje. Wszystko to jest częścią naturalnego postępu rozwoju jak najlepszego Języka, który przez długi czas będzie wywierał istotny wpływ na branżę. Rozwój odbywa się przez otwartą dyskusję, analizę istniejących dokumentów, „plemienną" wiedzę biznesową, która ostatecznie się tworzy, a także w wyniku odwoływania się do standardów, słowników i tezaurusów. Trzeba również pamiętać o sformułowaniach, w których pewne słowa i frazy nie pasują do kontekstu biznesowego tak dobrze, jak początkowo sądziliśmy; uświadamiamy sobie wtedy, że inne pasują dużo lepiej.

Zastanówmy się nad tym, w jaki sposób tworzy się ten niezwykle ważny Język Wszechobecny. Poniżej podano kilka sprawdzonych metod. Pamiętajmy, że eksperymentowanie prowadzi do postępów.

- Stwórz rysunki dziedziny fizycznej i pojęciowej i oznacz je etykietami opisującymi nazwy i akcje. Te rysunki są głównie nieformalne, ale mogą zawierać pewne aspekty formalnego modelowania oprogramowania. Nawet jeśli zespół stosuje pewne elementy formalnego modelowania z wykorzystaniem języka **UML** (ang. *Unified Modeling Language*), to staramy się unikać formalnego stylu, aby nie ograniczać dyskusji i nie usztywnić kreatywności szukanego Języka w ostatecznej formie.
- Utwórz glosariusz pojęć z prostymi definicjami. Wymień pojęcia alternatywne, włącznie z tymi, które są obiecujące, oraz tymi, które się nie sprawdziły. Zapisz informację o tym, jakie były powody niepowodzenia. Podczas dołączania definicji w gruncie rzeczy opracowujemy dla Języka frazy wielokrotnego użytku, ponieważ jesteśmy zmuszeni do pisania z wykorzystaniem Języka dziedziny.
- Jeżeli nie podoba Ci się pomysł glosariusza, posłuż się jakimś rodzajem dokumentacji, która zawiera nieformalne schematy ważnych pojęć oprogramowania. Tak jak powiedziałem wcześniej, celem jest znalezienie dodatkowych pojęć i fraz Języka.
- Ponieważ glosariusz lub inne pisane dokumenty mogły być utworzone przez jednego albo kilku członków zespołu, udostępnij je pozostałym członkom zespołu w celu wyrażenia opinii. Nie zawsze musisz się zgadzać ze wszystkimi podanymi terminami. Należy zatem zachować elastyczność i gotowość do redakcji.

Są to pewne idealne pierwsze kroki zmierzające do stworzenia Języka Wszechobecnego, które pasują do konkretnej dziedziny. Jednakże absolutnie nie jest to model, który rozwijamy. Jest to tylko początkowa postać Języka Wszechobecnego, która wkrótce zostanie wyrażona w kodzie źródłowym Twojego systemu. Mówimy w Javie, C#, Scali albo jakimś innym wybranym języku programowania. Te rysunki i dokumenty nie uwzględniają również tego, że Język Wszechobecny będzie się nadal rozwijał i przekształcał. Artefakty, które pierwotnie prowadziły nas po inspirującej ścieżce zmierzającej do rozwijania przydatnego Języka Wszechobecnego dopasowanego do określonej dziedziny specjalistycznej, z czasem staną się przestarzałe. *Właśnie dlatego to mowa zespołu i model w kodzie są najtrwalszym i jedynym gwarantowanym opisem Języka Wszechobecnego.*

Ponieważ mowa zespołu i model w kodzie są trwałym wyrażeniem Języka Wszechobecnego, należy przygotować się na wyeliminowanie rysunków, glosariuszy i innej dokumentacji. Trudno byłoby utrzymać tę dokumentację w zgodzie

z mówionym Językiem Wszechobecnym i kodem źródłowym, które są szybko usprawniane. Nie jest to wymaganie związane ze stosowaniem DDD, ale rozwiązanie pragmatyczne, gdyż utrzymywanie synchronizacji całej dokumentacji z systemem staje się niepraktyczne.

Dysponując tą wiedzą, możemy zmienić projekt przykładu `saveCustomer()`. Zastanówmy się, jak doprowadzić do tego, aby klasa `Customer` odzwierciedlała wszystkie możliwe cele biznesowe, które powinna wspierać.

```
public interface Customer {
    public void changePersonalName(
        String firstName, String lastName);
    public void postalAddress(PostalAddress postalAddress);
    public void relocateTo(PostalAddress changedPostalAddress);
    public void changeHomeTelephone(Telephone telephone);
    public void disconnectHomeTelephone();
    public void changeMobileTelephone(Telephone telephone);
    public void disconnectMobileTelephone();
    public void primaryEmailAddress(EmailAddress emailAddress);
    public void secondaryEmailAddress(EmailAddress emailAddress);
}
```

Można by spierać się, że to nie jest najlepszy model dla klasy `Customer`, ale podczas stosowania metod DDD kwestionowanie projektu jest oczekiwane. Zespół może swobodnie dyskutować na temat tego, co jest najlepszym modelem, i przyjąć ustalenia dopiero po odkryciu i uzgodnieniu Języka Wszechobecnego. Interfejs zaprezentowany powyżej jawnie odzwierciedla różne biznesowe cele, które powinna wspierać klasa `Customer` pomimo wielu ulepszeń Języka możliwych do wprowadzenia w przyszłości.

Trzeba również wziąć pod uwagę, że wzorzec Usługa Aplikacji także byłby refaktoryzowany tak, by odzwierciedlał wyraźnie cele biznesowe. Każda metoda Usługi Aplikacji byłaby modyfikowana w celu obsługi pojedynczego przypadku użycia albo historyjki użytkownika:

```
@Transactional
public void changeCustomerPersonalName(
    String customerId,
    String customerFirstName,
    String customerLastName) {

    Customer customer = customerRepository.customerOfId(customerId);

    if (customer == null) {
        throw new IllegalStateException("Customer does not exist.");
    }

    customer.changePersonalName(customerFirstName, customerLastName);
}
```

Pokazana metoda różni się od oryginalnego przykładu, w którym wykorzystano pojedynczą metodę do obsługi wielu różnych przypadków użycia lub historyjek użytkownika. W nowym przykładzie ograniczyliśmy pojedynczą metodę Usługi Aplikacji wyłącznie do zmiany nazwiska klienta. Metoda nie robi nic więcej. Zatem kiedy decydujemy się na zastosowanie DDD, naszym zadaniem jest odpowiednie udoskonalanie Usług Aplikacji. To sugeruje, że interfejs użytkownika odzwierciedla węższy zakres celu użytkownika niż ten, który został pierwotnie wyznaczony. Jednak teraz ta określona metoda Usługi Aplikacji nie wymaga od klienta podawania dziesięciu wartości `null` za parametrami opisującymi imię i nazwisko.

Czy ten nowy styl projektu ułatwia zadanie naszemu umysłowi? Możemy czytać kod i bez trudu go rozumiemy. Możemy też go przetestować, by potwierdzić, że robi dokładnie to, co powinien, oraz że nie robi niczego, czego robić nie powinien.

W ten sposób Język Wszechobecny staje się zespołowym wzorcem pozwalającym uchwycić pojęcia i terminy określonej głównej dziedziny biznesowej w samym modelu oprogramowania. Ten model oprogramowania obejmuje rzeczowniki, przymiotniki, czasowniki oraz bogatsze wyrażenia stworzone w sposób formalny i używane przez zespół. Zarówno oprogramowanie, jak i testy weryfikujące zgodność modelu z regułami dziedziny są wyrażone w Języku, którym posługuje się zespół.

### Wszechobecny, ale nie uniwersalny

Przydałoby się dodatkowe objaśnienie dotyczące zasięgu Języka Wszechobecnego. Jest kilka podstawowych pojęć, o których powinniśmy pamiętać:

- „Wszechobecny" oznacza „dominujący" albo „obecny wszędzie", *ponieważ Językiem mówią członkowie zespołu i wyraża go model dziedziny*, który zespół rozwija.
- Użycie słowa *wszechobecny* nie jest próbą zaznaczenia, że chodzi o rodzaj uniwersalnego języka dziedziny obowiązującego w całym przedsiębiorstwie lub na całym świecie.
- Jeden Język Wszechobecny obowiązuje w jednym Kontekście Ograniczonym.
- Konteksty Ograniczone są stosunkowo małe — mniejsze niż można by to sobie początkowo wyobrazić. Kontekst Ograniczony jest wystarczająco duży tylko do tego, aby uchwycić kompletny Język Wszechobecny odizolowanej dziedziny biznesowej, i ani trochę większy.
- Język jest wszechobecny tylko wewnątrz zespołu, który pracuje nad projektem rozwijającym odizolowany Kontekst Ograniczony.

- W projekcie, w którym jest rozwijany pojedynczy Kontekst Ograniczony, zawsze istnieje co najmniej jeden dodatkowy odizolowany Kontekst Ograniczony, z którym podstawowy kontekst się integruje, używając **Map Kontekstu (3)**. Każdy z wielu Kontekstów Ograniczonych integrujących się ze sobą ma swój własny Język Wszechobecny, chociaż mogą istnieć terminy, które częściowo się pokrywają.
- Próba zastosowania jednego Języka Wszechobecnego w całym przedsiębiorstwie albo, jeszcze gorzej, Języka uniwersalnego dla wielu przedsiębiorstw zakończy się niepowodzeniem.

Gdy zaczynasz nowy projekt i masz zamiar prawidłowo stosować metodykę DDD, zidentyfikuj odizolowany Kontekst Ograniczony, który będzie rozwijany. To wyznacza wyraźną granicę wokół Twojego modelu dziedziny. Należy omawiać, przeprowadzać badania, tworzyć pojęcia, rozwijać i mówić Językiem Wszechobecnym odizolowanego modelu dziedziny w ramach jawnie określonego Kontekstu Ograniczonego. Należy odrzucać wszystkie pojęcia, które nie są częścią uzgodnionego Języka Wszechobecnego Twojego odizolowanego Kontekstu.

## Wartość biznesowa używania technik DDD

Jeżeli doświadczenie czytelnika w jakimś stopniu przypomina moje, to z pewnością jest mu wiadome, że deweloperzy oprogramowania nie są w stanie używać wszystkich technologii i technik tylko dlatego, że wydają się one świetne albo intrygujące. Musimy uzasadnić wszystko, co robimy. Myślę, że nie zawsze tak było, ale dobrze, że taka zasada obowiązuje teraz. Uważam, że najlepszym uzasadnieniem stosowania dowolnej technologii lub techniki jest dostarczanie wartości dla biznesu. Jeżeli jesteśmy w stanie ustalić prawdziwą, namacalną wartość biznesową, to z jakich powodów biznes mógłby odrzucić to, co polecamy?

Przypadek biznesowy jest wzmocniony zwłaszcza wtedy, gdy możemy zademonstrować, że w porównaniu z innymi możliwościami wartości biznesowe wynikające z zastosowania polecanego przez nas podejścia są większe.

> **Czy wartość biznesowa nie jest najważniejsza?**
>
> Z pewnością jest. Być może wcześniej powinienem zastosować podtytuł „Wartość biznesowa używania DDD". Ale robię to teraz. Ten podtytuł mógłby właściwie brzmieć: „Jak sprzedawać DDD szefowi?". Do czasu, kiedy nie będziesz przekonany, że istnieje realna szansa implementacji DDD w Twojej firmie, ta książka jest tylko hipotetyczna. Nie chcę, aby lektura tej publikacji była dla czytelnika wyłącznie ćwiczeniem teoretycznym. Proponuję, by w trakcie czytania tej książki porównywać opisane w niej sytuacje do konkretnej rzeczywistości swojego przedsiębiorstwa. Wtedy można łatwiej zaobserwować realne korzyści, jakie może osiągnąć firma. Zachęcam zatem do kontynuowania lektury.

Rozważmy bardzo realistyczną biznesową wartość zastosowania DDD. Warto otwarcie podzielić się tymi spostrzeżeniami z kierownictwem, ekspertami dziedziny i technicznymi członkami zespołu. Poniżej sumarycznie przedstawiłem wartość i korzyści, które można osiągnąć. W dalszej części opisałem je bardziej szczegółowo. Zacznę od korzyści o mniej technicznym charakterze:

1. Organizacja zyskuje przydatny model swojej dziedziny.
2. Powstaje udoskonalona i dokładna definicja biznesu.
3. Eksperci dziedziny przyczyniają się do tworzenia projektu oprogramowania.
4. Użytkownicy zyskują system wygodniejszy do używania.
5. Wokół modeli tworzone są czytelne granice.
6. Architektura przedsiębiorstwa jest lepiej zorganizowana.
7. Stosowane jest zwinne, iteracyjne, ciągłe modelowanie.
8. Wykorzystywane są nowe narzędzia, zarówno na poziomie strategicznym, jak i taktycznym.

## 1. Organizacja zyskuje przydatny model swojej dziedziny

Metodologia DDD kładzie nacisk na zainwestowanie wysiłków w działania, które mają największe znaczenie dla biznesu. Nie staramy się stosować nadmiernego modelowania. Skupiamy się na Dziedzinie Głównej. Istnieją również inne modele, które wspierają Dziedzinę Główną. One także są ważne. Jednak modele pomocnicze nie zawsze otrzymują taki priorytet i nie zawsze poświęca się im tyle wysiłku co Dziedzinie Głównej.

Dzięki temu, że koncentrujemy się na tym, co odróżnia nasz biznes od wszystkich innych, możemy dobrze zrozumieć naszą misję i poznajemy parametry, które powinniśmy śledzić. Dostarczymy dokładnie to, co jest potrzebne do tego, by osiągnąć konkurencyjną przewagę.

## 2. Powstaje udoskonalona i dokładna definicja biznesu

Dzięki DDD biznes i jego misja mogą zostać zrozumiane lepiej niż kiedykolwiek wcześniej. Słyszałem kiedyś zdanie, zgodnie z którym Język Wszechobecny opracowany dla Dziedziny Głównej biznesu znalazł swoją drogę do materiałów marketingowych. Z pewnością należałoby go włączyć do dokumentów dotyczących wizji i misji.

W miarę upływu czasu i udoskonalania modelu w biznesie rozwija się głębokie zrozumienie problematyki biznesu — może to służyć za narzędzie analizy. Eksperci dziedziny ujawniają szczegóły, które są kształtowane przez technicz-

nych partnerów w zespole. Te szczegóły pomagają w analizie wartości bieżącego kierunku oraz kierunków w przyszłości — zarówno na poziomie strategicznym, jak i taktycznym.

## 3. Eksperci dziedziny przyczyniają się do tworzenia projektu oprogramowania

Zyskanie głębszego zrozumienia sedna biznesu przez członków organizacji tworzy wartość biznesową. Eksperci dziedziny nie zawsze zgadzają się w kwestii pojęć i terminologii. Czasami różnicom sprzyjają różne doświadczenia z zewnątrz, ukształtowane zanim eksperci dołączyli do organizacji. Innym razem różnice wynikają z rozbieżnych ścieżek obranych przez każdego eksperta w ramach tej samej organizacji. Pomimo różnic eksperci dziedziny zaangażowani w proces DDD osiągają konsensus. To wzmacnia siłę zespołu i organizację jako całość.

Deweloperzy zyskują teraz wspólny Język w ramach jednolitego zespołu — razem z ekspertami dziedziny. Dodatkowe korzyści osiągają z transferu wiedzy od ekspertów dziedziny, z którymi pracują. Łatwiejsze jest również szkolenie i przekazywanie wiedzy, co jest istotne ze względu na to, że deweloperzy często przychodzą i odchodzą — albo do nowej Dziedziny Głównej, albo w ogóle opuszczają organizację. Szanse na rozwój „wiedzy plemiennej", kiedy tylko nieliczni wybrańcy rozumieją model, są mniejsze. Eksperci, pozostali deweloperzy oraz nowi członkowie zespołu kontynuują dzielenie się wspólną wiedzą, dostępną dla wszystkich osób w organizacji, które jej potrzebują. Ta korzyść jest możliwa do osiągnięcia ze względu na wyraźny cel — zachowanie Języka dziedziny.

## 4. Użytkownicy zyskują system wygodniejszy do używania

Często można poprawić wygodę korzystania użytkowników z systemu dzięki lepszemu odzwierciedleniu modelu dziedziny. Projektowanie DDD ma formalnie „wrodzoną" cechę wywierania wpływu na sposób posługiwania się oprogramowaniem przez użytkowników.

Gdy oprogramowanie pozostawia swoim użytkownikom zbyt dużo do rozumienia, powstaje konieczność szkolenia użytkowników w celu uzyskania zdolności podejmowania większości decyzji. W gruncie rzeczy użytkownicy przenoszą zrozumienie tematu ze swoich umysłów na dane wprowadzane w formularzach. Dane te są następnie składowane w magazynie danych. Jeżeli użytkownicy nie rozumieją dokładnie, co jest potrzebne, wyniki są nieprawidłowe. To często prowadzi do zgadywanki, która kończy się obniżoną produktywnością, trwającą do czasu, aż użytkownicy zrozumieją oprogramowanie.

Gdy wrażenia użytkownika zostaną zaprojektowane tak, by były zgodne z modelem stworzonym przez ekspertów dziedziny, użytkownicy mogą wyciągać poprawne wnioski. Oprogramowanie w istocie szkoli użytkowników, co zmniejsza

koszty szkolenia dla biznesu. Szybsza droga do produktywności z mniejszą liczbą szkoleń — to jest wartość biznesowa.

W kolejnych punktach przejdziemy do bardziej technicznych korzyści dla biznesu.

## 5. Wokół modeli tworzone są czytelne granice

Zespół techniczny w mniejszym stopniu zajmuje się działaniami, które mogą wydawać się im użyteczne z punktu widzenia algorytmów i oprogramowania, ponieważ musi sprostać oczekiwaniom w postaci korzyści dla biznesu. Czystość kierunku pozwala się skupić na sile rozwiązania — skierować wysiłek tam, gdzie ma on największe znaczenie. Osiągnięcie tego celu jest bardzo ściśle powiązane ze zrozumieniem Kontekstu Ograniczonego projektu.

## 6. Architektura przedsiębiorstwa jest lepiej zorganizowana

Gdy Konteksty Ograniczone są dobrze rozumiane i uważnie podzielone, wszystkie zespoły w przedsiębiorstwie zaczynają dokładnie rozumieć, gdzie i dlaczego są potrzebne integracje. Zarówno granice, jak i relacje pomiędzy nimi są jasno wydzielone. Zespoły, które stosują przecinające się modele poprzez zależności korzystania, używają Map Kontekstu w celu ustalenia formalnych relacji i sposobów integracji. W rezultacie może to doprowadzić do bardzo gruntownego zrozumienia całej architektury przedsiębiorstwa.

## 7. Stosowane jest zwinne, iteracyjne, ciągłe modelowanie

Słowo *projektowanie* może wywoływać negatywne skojarzenia w umysłach przedstawicieli kierownictwa. DDD nie jest jednak trudnym, mocno celebrowanym procesem projektowania i rozwoju. DDD nie polega na rysowaniu wykresów. Polega na uważnym dostrajaniu modelu myślenia ekspertów dziedziny i przekształcaniu go do postaci modelu przydatnego dla biznesu. Nie chodzi o stworzenie modelu z rzeczywistego świata, tak jak w modelach, które próbują naśladować rzeczywistość.

Wysiłki zespołu są podejmowane zgodnie ze zwinnym podejściem do projektowania — bazującym na procesie iteracyjnym i przyrostowym. W projekcie DDD może być skutecznie zastosowany dowolny proces, którym zespół sprawnie się posługuje. Stworzonym modelem jest działające oprogramowanie. Jest ono ciągle udoskonalane — tak długo, aż biznes przestanie go potrzebować.

### 8. Wykorzystywane są nowe narzędzia, zarówno na poziomie strategicznym, jak i taktycznym

Kontekst Ograniczony wyznacza zespołowi granicę modelu, w obrębie której można utworzyć rozwiązanie określonej biznesowej dziedziny problemu. Wewnątrz pojedynczego Kontekstu Ograniczonego wykorzystywany jest Język Wszechobecny sformułowany przez zespół. Posługują się nim członkowie zespołu i jest on obecny w modelu oprogramowania. Zasadniczo odmienne zespoły (czasami każdy odpowiedzialny jest za inny Kontekst Ograniczony) używają Map Kontekstu w celu strategicznej segregacji Kontekstów Ograniczonych i zrozumienia ich integracji. W obrębie pojedynczej granicy modelowania zespół może wykorzystać dowolną liczbę przydatnych narzędzi modelowania: *Agregaty* (10), *Encje* (5), *Obiekty Wartości* (6), *Usługi* (7), *Zdarzenia Dziedziny* (8) i inne.

## Wyzwania związane ze stosowaniem DDD

Podczas wdrażania DDD napotykamy rozmaite wyzwania. Jest to typowe dla każdego, komu udało się skutecznie wdrożyć to rozwiązanie. Jakie są popularne wyzwania i jak można usprawiedliwić użycie DDD, gdy przed nimi staniemy? Poniżej omówię najpopularniejsze z nich:

- koszty czasu i pracy, jakie trzeba ponieść w związku z tworzeniem Języka Wszechobecnego;
- zaangażowanie ekspertów dziedziny w początkowej fazie projektu i przez cały czas jego trwania;
- zmiana sposobu myślenia deweloperów o rozwiązaniach w ich dziedzinie.

Jednym z największych wyzwań w używaniu DDD są koszty czasu i pracy, jakie trzeba ponieść podczas myślenia o dziedzinie biznesu, badaniu pojęć i terminologii oraz rozmowach z ekspertami dziedziny w celu odkrywania, ujednolicania i ulepszania Języka Wszechobecnego. Nie można od razu przejść do kodowania w technopaplaninie. Jeżeli chcemy zastosować DDD w sposób kompletny, tak aby przyniosło to największą wartość dla biznesu, trzeba włożyć w to więcej energii umysłowej i wysiłku, co wymaga więcej czasu. Na wszystko jest potrzebny czas.

Wyzwaniem może być także utrzymanie koniecznego zaangażowania ekspertów dziedziny. Niezależnie od tego, jak trudno to osiągnąć, jest to konieczne. Jeżeli w projekt nie zaangażuje się co najmniej jeden prawdziwy ekspert, nie uda nam się odkryć głębokiej wiedzy z zakresu dziedziny. Gdy uda nam się zdobyć zaangażowanie ekspertów dziedziny, możemy skoncentrować się na deweloperach. Deweloperzy muszą ze sobą rozmawiać i uważnie słuchać prawdziwych ekspertów, modelując język mówiony na oprogramowanie, które odzwierciedla ich mentalny model dziedziny.

Jeżeli dziedzina, w której pracujemy, jest naprawdę specyficzna dla naszego biznesu, to eksperci dziedziny mają najbardziej kluczową wiedzę zamkniętą w swoich umysłach. Trzeba ją stamtąd wyciągnąć. Uczestniczyłem w projektach, w przypadku których spotkania z ekspertami dziedziny były bardzo utrudnione. Czasami eksperci dużo podróżowali. Bywało, że pomiędzy jednogodzinnymi spotkaniami upływało kilka tygodni. W małych przedsiębiorstwach ekspertem dziedziny może być dyrektor zarządzający albo jeden z wiceprezesów. Osoby te mają na głowie wiele innych spraw, które mogą im się wydawać ważniejsze.

### Logika kowboja

AJ: Jeżeli nie potrafisz schwytać dużego bawoła, będziesz musiał głodować.

Zdobycie zaangażowania ekspertów dziedziny może wymagać kreatywności…

### Jak zaangażować ekspertów dziedziny w projekt

Przerwa na kawę. Używaj takiego Języka Wszechobecnego:

„Cześć, Zosiu, przyniosłem ci duży kubek lekkiej, ekstragorącej, mieniącej się latte z pianką. Czy znajdziesz kilka minut, by porozmawiać…?”.

Naucz się używać Języka Wszechobecnego kierownictwa poziomu C: „…zyski, …, …dochody, …przewaga konkurencyjna, …dominacja rynku”. Naprawdę warto.

Notatki na serwetce.

Większość deweloperów musi *zmienić sposób myślenia*, by móc skutecznie zastosować DDD. My, deweloperzy, myślimy w kategoriach technicznych. Techniczne rozwiązania przychodzą nam łatwo. Nie chcę powiedzieć, że myślenie technicznie jest czymś złym. Chodzi o to, że czasami myślenie w sposób mniej techniczny jest lepsze. Jeżeli przez całe lata mieliśmy w zwyczaju praktykować rozwijanie oprogramowania wyłącznie w sposób techniczny, to być może nadszedł odpowiedni czas, aby zastanowić się nad nowym sposobem myślenia. Rozwijanie Języka Wszechobecnego swojej dziedziny jest najlepszym punktem wyjścia.

### Logika kowboja

LB: Buty tego faceta są za małe. Jeżeli nie znajdzie sobie innej pary, zaczną boleć go palce.

AJ: Tak jest. Jeśli nie będziesz słuchać, będziesz musiał poczuć.


W przypadku stosowania DDD potrzebny jest inny poziom myślenia, wykraczający poza nazewnictwo pojęć. Gdy modelujemy dziedzinę poprzez oprogramowanie, musimy dokładnie zastanowić się nad tym, co robią poszczególne obiekty modelu. Modelowanie sprowadza się do *projektowania zachowania obiektów*. To prawda. Chcemy, aby zachowania były właściwie nazwane, tak by oddawały istotę Języka Wszechobecnego. Ale trzeba także wziąć pod uwagę to, co robi obiekt, w kontekście określonego zachowania. Ten wysiłek wykracza poza tworzenie atrybutów klasy i wystawianie publicznych getterów i setterów na użytek klientów modelu.

Przyjrzyjmy się teraz bardziej interesującej dziedzinie — takiej, która stawia więcej wyzwań w porównaniu z prostą dziedziną omawianą wcześniej. Celowo powtarzam moje poprzednie wskazówki, aby utrwalić pojęcia.

Zastanówmy się, co się stanie, jeżeli ograniczymy się do przygotowania procedur dostępu do danych naszego modelu. Mówiąc dokładniej, jeśli udostępnimy same procedury dostępu do danych obiektów modelu, w rezultacie uzyskamy jedynie model danych. Przyjrzyjmy się dwóm przykładom zaprezentowanym poniżej i zastanówmy się, który z nich wymaga dokładniejszej analizy projektowej oraz który przynosi większą korzyść dla klientów. Wymaganie dotyczy modelu Scrum. Polega na przypisaniu pozycji rejestru zadań do wykonania do sprintu. Prawdopodobnie robisz to cały czas, więc jest to dla Ciebie dość znajoma dziedzina.

W pierwszym przykładzie, tak jak powszechnie się to robi, wykorzystano metody dostępu do atrybutów:

```
public class BacklogItem extends Entity {
    private SprintId sprintId;
    private BacklogItemStatusType status;
    ...
    public void setSprintId(SprintId sprintId) {
        this.sprintId = sprintId;
    }

    public void setStatus(BacklogItemStatusType status) {
        this.status = status;
    }
    ...
}
```

A oto kod klienta tego modelu:

```
// klient przypisuje pozycję rejestru do sprintu
// poprzez ustawienie atrybutów sprintId i status

backlogItem.setSprintId(sprintId);
backlogItem.setStatus(BacklogItemStatusType.COMMITTED);
```

W drugim przykładzie skorzystano z zachowania obiektu dziedziny, które wyraża Język Wszechobecny dziedziny:

```
public class BacklogItem extends Entity {
    private SprintId sprintId;
    private BacklogItemStatusType status;
    ...

    public void commitTo(Sprint aSprint) {
        if (!this.isScheduledForRelease()) {
            throw new IllegalStateException(
                "Pozycja musi być zaplanowana do wydania, aby można ją było przypisać
                ↪do sprintu".);
        }

        if (this.isCommittedToSprint()) {
            if (!aSprint.sprintId().equals(this.sprintId())) {
                this.uncommitFromSprint();
            }
        }

        this.elevateStatusWith(BacklogItemStatus.COMMITTED);

        this.setSprintId(aSprint.sprintId());

        DomainEventPublisher
           .instance()
           .publish(new BacklogItemCommitted(
                   this.tenant(),
                   this.backlogItemId(),
                   this.sprintId()));
    }
    ...
}
```

Wydaje się, że klient tego wyraźnego modelu stąpa po bezpieczniejszym gruncie:

```
// klient przypisuje pozycję rejestru do sprintu,
// przejawiając zachowanie specyficzne dla dziedziny

backlogItem.commitTo(sprint);
```

W pierwszym przykładzie skorzystano z podejścia bardzo ukierunkowanego na dane. Obciążenie związane z poprawnym przypisaniem pozycji rejestru do

sprintu w całości spoczywa na kliencie. Model, który w istocie nie jest modelem dziedziny, zupełnie w tym nie pomaga. Co się stanie, jeżeli klient przez pomyłkę zmieni tylko pole `sprintId`, a nie zmieni pola `status` lub postąpi odwrotnie? Albo co się stanie, jeżeli w przyszłości zajdzie potrzeba ustawienia wartości innego atrybutu? Trzeba będzie przeanalizować kod klienta, aby zaimplementować prawidłowe odwzorowanie wartości danych na odpowiednie atrybuty obiektu `BacklogItem`.

Zastosowanie tego podejścia ujawnia również kształt obiektu `BacklogItem` i wyraźnie skupia uwagę na jego atrybutach danych, a nie na jego zachowaniach. Nawet jeśli przyjęlibyśmy, że metody `setSprintId()` i `setStatus()` są zachowaniami, to te „zachowania" nie przynoszą żadnej prawdziwej biznesowej wartości dla dziedziny. Te „zachowania" nie wskazują jawnie zamiarów scenariuszy, które oprogramowanie dziedziny powinno modelować — tzn. przypisywania pozycji rejestru do sprintu. Powodują one kognitywne przeciążenie, gdy deweloper klienta próbuje w myślach wybrać te spośród atrybutów obiektu `BacklogItem`, które są wymagane do przypisania pozycji do sprintu. Takich atrybutów może być dużo, ponieważ jest to model ukierunkowany na dane.

Rozważmy teraz drugi przykład. Zamiast udostępniania klientom atrybutów danych model udostępnia zachowanie, które jawnie i wyraźnie wskazuje, że klient może przypisać pozycję rejestru do sprintu. Eksperci w tej konkretnej dziedzinie dyskutują następujące wymaganie modelu:

> Każdą pozycję rejestru można przypisać do sprintu. Pozycję można przypisać do sprintu tylko wtedy, gdy została zaplanowana do wydania. Jeżeli została już przypisana do innego sprintu, to przypisanie musi być wcześniej anulowane. Gdy przypisanie zostanie zakończone, należy powiadomić o tym zainteresowanych.

Zatem metoda z drugiego przykładu oddaje Język Wszechobecny modelu w kontekście — tzn. Kontekście Ograniczonym, w którym wyizolowano typ `BacklogItem`. Co więcej, podczas analizy tego scenariusza odkryjemy, że pierwsze rozwiązanie jest niekompletne i zawiera błędy.

W przypadku zastosowania drugiej implementacji klienci nie muszą wiedzieć, co jest wymagane do realizacji przypisania — niezależnie od tego, czy jest to operacja prosta, czy złożona. W implementacji tej metody jest tylko tyle logiki, ile jest konieczne. Z łatwością dodaliśmy warunek zabezpieczający przed przypisaniem do sprintu pozycji rejestru, która jeszcze nie została zaplanowana do wydania. Oczywiście można także umieścić warunki wewnątrz setterów w pierwszej implementacji, ale settery będą odtąd odpowiedzialne za zrozumienie pełnego kontekstu stanu obiektu, a nie samych wymagań dotyczących atrybutów `sprintId` i `status`.

Istnieje również inna subtelna różnica. Zwróćmy uwagę, że jeżeli pozycja rejestru została wcześniej przypisana do innego sprintu, to najpierw będzie anulowane przypisanie jej aktualnego sprintu. To jest ważny szczegół, bo kiedy następuje anulowanie przypisania pozycji rejestru ze sprintu, trzeba zgłosić klientom Zdarzenie Dziedziny:

> Można anulować przypisanie do sprintu każdej pozycji rejestru. W przypadku anulowania przypisania pozycji rejestru do sprintu należy powiadomić zainteresowanych.

Opublikowanie powiadomienia o anulowaniu przypisania uzyskuje się „za darmo" — poprzez użycie zachowania dziedziny `uncommitFrom()`. Metoda `commitTo()` nie musi nawet wiedzieć o tym, że inicjuje powiadomienie. Musi jedynie wiedzieć, że przed przypisaniem pozycji rejestru do nowego sprintu trzeba anulować przypisanie tej pozycji rejestru do dowolnych innych sprintów. Dodatkowo w ramach zachowania dziedziny `commitTo()` następuje powiadomienie za pomocą Zdarzenia zainteresowanych stron. Bez umieszczenia tego bogatego zachowania w obrębie obiektu `BacklogItem` trzeba by było publikować Zdarzenia z poziomu klientów. To z pewnością spowodowałoby ujawnienie logiki dziedziny z modelu. To nie jest dobre.

Wyraźnie widać, że stworzenie obiektu `BacklogItem` zgodnego z drugim przykładem wymaga więcej przemyśleń w porównaniu z pierwszym. Chociaż wysiłek intelektualny nie jest znacząco większy, korzyści są duże. Im więcej uczymy się projektować w ten sposób, tym projektowanie staje się łatwiejsze. Ostatecznie na pewno trzeba włożyć więcej wysiłku intelektualnego i więcej pracy, nasilić współpracę i zgrywanie wysiłków zespołu, ale nie do tego stopnia, aby metodyka DDD stała się złożonym procesem. Nowy sposób myślenia jest wart wkładanego w to wysiłku.

**Zapiski na tablicy**

- Używając konkretnej dziedziny, w której aktualnie pracujesz, wyszukaj popularne pojęcia i akcje w modelu.
- Zapisz pojęcia na tablicy.
- Następnie zapisz frazy, które powinny być używane przez zespół podczas rozmów dotyczących projektu.
- Podyskutuj o nich z prawdziwym ekspertem dziedziny, by dowiedzieć się, jak można je udoskonalić (pamiętaj, żeby przynieść kawę).

### *Uzasadnienie dla modelowania dziedziny*

*Modelowanie taktyczne* jest, ogólnie rzecz biorąc, bardziej złożone od *modelowania strategicznego*. Zatem jeżeli zamierzasz rozwijać model dziedziny używający taktycznych wzorców DDD (Agregaty, Usługi, Obiekty Wartości, Zdarzenia itd.), będziesz zmuszony do wykonania większej pracy umysłowej i poniesienia większych nakładów finansowych. Jeśli tak jest, to w jaki sposób organizacja uzasadnia taktyczne modelowanie dziedziny? Jakich kryteriów należy użyć, by zakwalifikować określony projekt do dodatkowych inwestycji niezbędnych do prawidłowego zastosowania DDD od góry do dołu?

Wyobraź sobie, że przewodzisz wyprawie po nieznanym terenie. Chciałbyś poznać okoliczne lądy i granice. Twój zespół studiuje mapy, być może nawet rysuje swoje własne, i określa swoje strategiczne podejście. Masz zamiar przeanalizować elementy terenu i zastanowić się, jak można by ich użyć, aby przyniosły korzyść ekspedycji. Niezależnie od wysiłków wkładanych w planowanie niektóre aspekty takiego przedsięwzięcia będą naprawdę trudne.

Jeżeli z Twojej strategii wynika, że musisz się wspiąć na pionowe zbocze skały, to do takiej wspinaczki potrzebujesz odpowiednich narzędzi taktycznych i właściwych posunięć. Stojąc na dole i patrząc w górę, można zauważyć pewne sygnały określonych wyzwań i niebezpiecznych obszarów. Pomimo to nie wszystkie szczegóły będą widoczne. Zobaczysz je dopiero wtedy, kiedy zaczniesz się wspinać. Być może będziesz zmuszony do zainstalowania haków na gładkiej skale, ale prawdopodobne jest również, że wystarczą kliny w różnych rozmiarach, instalowane w naturalnych pęknięciach. Aby móc przypiąć się do tego zabezpieczenia, będziesz potrzebował zapięć typu karabińczyk. Możesz spróbować obrać jak najprostszą ścieżkę, ale musisz dokonać szczegółowych ustaleń — punkt po punkcie. Czasami może nawet wystąpić konieczność wycofania się i zmiany trasy, w zależności od tego, co „podyktuje" skała. Wiele osób uważa wspinaczkę za niebezpieczny sport, jednak te osoby, które rzeczywiście się wspinają, mówią, że to jest bezpieczniejsze niż jazda samochodem albo latanie samolotem. Wydaje się oczywiste, że aby tak było, prawdziwi alpiniści muszą rozumieć narzędzia i techniki pozwalające na właściwą ocenę skały.

Jeżeli rozwijanie określonej **Poddziedziny (2)** wymaga takiej trudnej, a nawet niepewnej wspinaczki, to warto zastosować taktyczne wzorce DDD. W inicjatywie biznesowej, która pasuje do kryteriów Dziedziny Głównej, nie należy zbyt szybko rezygnować z użycia wzorców taktycznych. Dziedzina Główna jest nieznanym i skomplikowanym obszarem. Najlepszą ochronę przed katastrofalnym upadkiem podczas „wspinaczki" zespół uzyskuje dzięki zastosowaniu odpowiedniej taktyki.

Oto kilka praktycznych wskazówek. Zacznę od najbardziej ogólnych, a następnie przejdę do szczegółów.

- Jeżeli Kontekst Ograniczony jest rozwijany jako Dziedzina Główna, to ma znaczenie strategiczne dla sukcesu przedsięwzięcia. Podstawowy model nie jest dobrze rozumiany. Wymaga wielu eksperymentów i refaktoryzacji. Prawdopodobnie zasługuje na długotrwałe poświęcenie i ciągłe ulepszanie. Nie zawsze musi to być Dziedzina Główna. Niemniej jednak, jeżeli Kontekst Ograniczony jest złożony, nowatorski i będzie musiał przetrwać przez długi czas w trakcie wprowadzania zmian, należy rozważyć użycie wzorców taktycznych jako inwestycji w przyszłość Twojego biznesu. Zakładamy tutaj, że Dziedzina Główna zasługuje na najlepsze zasoby deweloperów o wysokich kwalifikacjach.
- Dziedzina, która może się stać **Poddziedziną Generyczną (2)** albo Poddziedziną Pomocniczą dla swoich konsumentów, może w gruncie rzeczy stać się

Dziedziną Główną dla biznesu. Nie zawsze oceniamy dziedzinę z punktu widzenia jej ostatecznych konsumentów. Jeżeli rozwijasz Kontekst Ograniczony jako główną inicjatywę biznesową, to jest to Twoja Dziedzina Główna niezależnie od tego, jak postrzegają ją klienci spoza biznesu. Należy się poważnie zastanowić nad użyciem wzorców taktycznych.

- Jeśli rozwijasz Poddziedzinę Pomocniczą, która z różnych powodów nie może być pozyskana z zewnątrz jako Poddziedzina Generyczna, to istnieje możliwość, że zastosowanie wzorców taktycznych przyniesie korzyść. W takim przypadku warto przeanalizować umiejętności zespołu i zastanowić się, czy model jest nowy i nowatorski. Można go uznać za nowatorski, jeżeli dodaje określoną wartość biznesową i obejmuje wiedzę specjalistyczną, a nie wtedy, gdy jest jedynie intrygujący z technicznego punktu widzenia. Jeśli zespół potrafi właściwie zastosować projekt taktyczny, a Poddziedzina Pomocnicza jest nowatorska i będzie musiała przetrwać przez wiele lat, to warto zainwestować w oprogramowanie, wykorzystując projektowanie taktyczne. Jednakże nie czyni to z tego modelu Dziedziny Głównej, ponieważ z punktu widzenia biznesu jest do jedynie Poddziedzina Pomocnicza.

Powyższe wytyczne mogą trochę ograniczać — zwłaszcza jeśli w Twojej firmie jest zatrudnionych wielu deweloperów z dużym doświadczeniem oraz wysokim poziomem umiejętności w zakresie modelowania dziedziny. Jeżeli doświadczenie jest bardzo duże i sami inżynierowie twierdzą, że wzorce taktyczne są najlepszym wyborem, to zaufanie ich opiniom ma sens. Uczciwi deweloperzy, niezależnie od tego, jak doświadczeni, w określonej sytuacji powiedzą, czy rozwijanie modelu dziedziny jest, czy nie jest najlepszym wyborem.

Typ samej dziedziny biznesu nie jest automatycznie czynnikiem decydującym o wyborze podejścia do projektowania. Zespół powinien wziąć pod uwagę ważne kwestie, które pomogą w podjęciu ostatecznej decyzji. Poniżej zamieszczono krótką listę bardziej szczegółowych parametrów decyzyjnych, które w mniejszym bądź w większym stopniu są dopasowane do wymienionych wcześniej bardziej ogólnych wskazówek.

- Czy eksperci dziedziny są dostępni i czy jesteś przekonany o słuszności idei tworzenia zespołu skupionego wokół nich?
- Czy złożoność określonej dziedziny biznesu, która jest dziś stosunkowo niska, z czasem wzrośnie? Istnieje ryzyko stosowania Skryptu Transakcji[1] w odniesieniu do złożonych aplikacji. Jeżeli używasz Skryptu Transakcji teraz, to czy wtedy, gdy Kontekst stanie się złożony, potencjalna refaktoryzacja do behawioralnego modelu dziedziny będzie wykonalna?

[1] W tym opisie uogólniłem pojęcia. Na tej liście użyłem terminu „Skrypt Transakcji" do reprezentacji podejść innych niż modelowanie dziedziny.

- Czy użycie taktycznych wzorców DDD ułatwi integrację z innymi Kontekstami Ograniczonymi — zarówno nabytymi na zewnątrz, jak i rozwijanymi samodzielnie?
- Czy w przypadku użycia wzorca Skrypt Transakcji rozwijanie oprogramowania naprawdę będzie prostsze i będzie wymagało mniej kodu? (Doświadczenie z obydwoma podejściami udowadnia, że w wielu przypadkach stosowanie Skryptu Transakcji wymaga takiej samej albo większej ilości kodu. To prawdopodobnie wynika z niedostatecznie dobrego zrozumienia złożoności dziedziny i innowacyjności modelu w fazie planowania projektu. Niedocenianie złożoności dziedziny i jej innowacyjności zdarza się naprawdę często).
- Czy kluczowa ścieżka i plan czasowy pozwalają na jakikolwiek narzut wymagany dla inwestycji taktycznej?
- Czy taktyczna inwestycja w Dziedzinę Główną ochroni system przed wpływami zmian w architekturze? Zastosowanie Skryptu Transakcji może narazić system na takie zmiany (modele dziedziny często są trwałe, architektura natomiast może wywierać negatywny wpływ na inne warstwy).
- Czy klienci (użytkownicy) skorzystają z czystszego, trwalszego podejścia do projektowania, czy też ich aplikację będzie można niedługo zastąpić gotowym rozwiązaniem? Inaczej mówiąc, czy istnieją jakiekolwiek powody, aby rozwijać produkt jako własną aplikację (usługę)?
- Czy rozwijanie aplikacji (usługi) z wykorzystaniem taktycznych wzorców DDD będzie trudniejsze niż skorzystanie z innych podejść — na przykład Skryptu Transakcji (kluczowe znaczenie dla udzielenia odpowiedzi na to pytanie mają poziom umiejętności i dostępność ekspertów dziedziny)?
- Gdybyśmy dysponowali kompletnym zestawem narzędzi do tego, by móc zastosować DDD, to czy zdecydowalibyśmy się na zastosowanie jakiegoś innego podejścia (niektóre czynniki sprawiają, że praktyczny staje się określony model utrwalania, na przykład mapowanie obiektowo-relacyjne, pełna serializacja Agregatów i utrwalanie, Magazyn Zdarzeń lub framework wspierający taktyczne wzorce DDD; mogą istnieć także inne czynniki)?

Ta lista nie została sporządzona z myślą o Twojej dziedzinie. Prawdopodobnie zdołałbyś podać kilka dodatkowych kryteriów. Zdajesz sobie sprawę z przyczyn stosowania najlepszych i najpotężniejszych metod pozwalających na osiągnięcie jak największych korzyści. Znasz także swoją branżę i masz obraz używanych technologii. W efekcie końcowym to klient biznesowy ma być zadowolony, a nie użytkownicy obiektów i technicy. Należy dokonywać mądrych wyborów.

### *DDD nie jest złożonym procesem*

W żadnym razie nie staram się sugerować, że prawidłowe stosowanie DDD prowadzi do „ciężkiego" procesu, obfitującego w skomplikowane procedury i obszerną dokumentację, którą trzeba utrzymywać. Metodyka DDD taka nie jest. Powinna ona dobrze pasować do dowolnego frameworka projektowania Agile, którego zespół używa, takiego jak Scrum. Jego zasady projektowania koncentrują się raczej na szybkich udoskonaleniach rzeczywistego modelu oprogramowania zgodnie z podejściem „najpierw test". W przypadku konieczności opracowania nowego obiektu dziedziny, na przykład Encji albo Obiektu Wartości, należy zastosować podejście „najpierw test", które opisano poniżej.

1. Napisz test, który demonstruje sposób użycia nowego obiektu dziedziny przez klienta modelu dziedziny.
2. Stwórz nowy obiekt dziedziny z kodem, który wystarcza do tego, aby test się skompilował.
3. Refaktoryzuj zarówno test, jak i obiekt dziedziny tak długo, aż test zacznie właściwie reprezentować sposób, w jaki klient powinien używać obiektu, a obiekt dziedziny będzie miał prawidłowe sygnatury metod reprezentujące zachowania.
4. Implementuj poszczególne zachowania obiektu tak długo, aż test zacznie przechodzić. Jednocześnie refaktoryzuj obiekt dziedziny do czasu, aż zostaną wyeliminowane wszelkie niewłaściwe duplikaty w kodzie.
5. Zademonstruj kod członkom zespołu, włącznie z ekspertami dziedziny, aby zyskać pewność, że test używa obiektu dziedziny zgodnie z aktualnym znaczeniem Języka Wszechobecnego.

Można by sądzić, że powyższy proces nie różni się w żaden sposób od podejścia „najpierw test", które stosujemy na co dzień. Być może jest ono trochę inne, ale w gruncie rzeczy jest takie samo. Faza testu nie usiłuje udowodnić z absolutną pewnością, że model jest „kuloodporny". W tym celu później zostaną dodane kolejne testy. Najpierw chcemy skupić się na tym, jak model będzie używany przez klientów. Te testy sterują projektem modelu. Dobrą wiadomością jest to, że na tym w gruncie rzeczy polega zwinne podejście do projektowania. Metodologia DDD promuje lekki rozwój, a nie ceremonialny, zawiły projekt „z góry". Pod tym względem to podejście niczym się nie różni od powszechnych technologii Agile. Zatem o ile nie sądzę, aby poprzednio opisane kroki oświeciły czytelnika w zakresie metodologii Agile, o tyle uważam, że wyjaśniły pozycję metod DDD, które powinny być używane zgodnie z zasadami Agile.

Na dalszym etapie projektu dodamy testy, które zweryfikują poprawność nowego obiektu dziedziny z każdej możliwej (i praktycznej) perspektywy. W tym momencie interesuje nas jedynie poprawność wyrażenia pojęcia dziedziny, które

jest ucieleśnione w nowym obiekcie. Czytanie demonstracyjnego kodu testu w stylu klienta musi ujawnić właściwą wyrazistość używania Języka Wszechobecnego. Eksperci dziedziny, którzy nie są „techniczni", powinni móc czytać kod (z pomocą dewelopera) na tyle dobrze, aby uzyskać jasne wrażenie, że model osiągnął cel, jaki zespół sobie postawił. To sugeruje, że dane testu muszą być realistyczne oraz muszą wspierać i poprawiać pożądaną ekspresywność. W przeciwnym razie eksperci dziedziny nie będą w stanie obiektywnie ocenić implementacji.

Pokazaną metodę zwinną „najpierw test" należy powtarzać tak długo, aż uzyskamy model, który działa zgodnie z zadaniami wyznaczonymi dla bieżącej iteracji. Kroki wymienione wcześniej są zgodne ze zwinnymi zasadami programowania. Reprezentują reguły pierwotnie promowane przez zwolenników Programowania Ekstremalnego. Stosowanie zwinnych reguł projektowania nie eliminuje żadnych zasadniczych wzorców i praktyk DDD. One nieźle do siebie pasują. Oczywiście można korzystać z pełnej metodologii DDD bez stosowania zasad projektowania „najpierw test". Zawsze można opracować testy dla istniejących obiektów modelu. Trzeba jednak pamiętać, że projektowanie z perspektywy klienta modelu dodaje bardzo pożądanego wymiaru.

## Fikcja z dużą dawką realizmu

Podczas rozważania na temat sposobu jak najlepszej prezentacji przewodnika właściwej implementacji DDD starałem się dostarczyć uzasadnienia dla wszystkiego, co zgodnie z tym, co powiedziałem, powinno być zrobione. To oznaczało dostarczanie odpowiedzi nie tylko na pytanie „jak?", ale także „dlaczego?". Przyszło mi na myśl, że przyjrzenie się kilku projektom jako studiom przypadków właściwie zilustruje powody, dla których coś zasugerowałem oraz pokazałem, że właściwe użycie DDD sprosta powszechnie występującym wyzwaniem.

Czasami łatwiej jest spojrzeć na problemy, jakie napotykają inne zespoły projektowe, i uczyć się na podstawie ich niewłaściwego używania DDD, niż szukać problemów we własnym projekcie. Jeśli uda nam się rozpoznać niedociągnięcia w pracy innych, z pewnością będziemy również w stanie ocenić, czy podążamy w tym samym, niepewnym kierunku albo nawet tkwimy po uszy w takim samym bagnie. Wiedząc, dokąd zmierzamy lub gdzie już jesteśmy, możemy wykonać precyzyjne korekty w celu wyeliminowania problemów i uniknięcia powtórzenia takich samych kłopotów w przyszłości.

Zamiast prezentować serię rzeczywistych projektów, nad którymi pracowałem — i tak nie mógłbym otwarcie o nich mówić — zdecydowałem się użyć trochę fikcji bazującej na realnych sytuacjach; przytrafiły się one zarówno mnie samemu, jak i innym. W ten sposób zdołałem utworzyć doskonały stan wydarzeń pozwalających na zademonstrowanie powodów, dla których określone podejście

do implementacji sprawdza się najlepiej albo przynajmniej pozwala najlepiej sprostać wyzwaniom towarzyszącym stosowaniu DDD.

Zatem studia przypadków nie opierają się wyłącznie na fikcji. Zaprezentowano w nich fikcyjne przedsiębiorstwo z prawdziwej branży, fikcyjne zespoły w przedsiębiorstwie z realnym oprogramowaniem do zbudowania i wdrożenia oraz rzeczywiste wyzwania i wynikające z nich problemy razem z realnymi rozwiązaniami. Nazwałem to fikcją z dużą dawką realizmu. Pisanie w tym stylu wydaje mi się całkiem skuteczne. Mam nadzieję, że czytelnik na tym skorzysta.

Podczas prezentowania dowolnego zbioru przykładów trzeba ograniczyć jego zakres, by opis był praktyczny. W przeciwnym razie, z powodu nadmiernej objętości, zarówno nauczanie, jak i uczenie się będą utrudnione. Przykłady nie mogą być także nadmiernie uproszczone, ponieważ to powodowałoby pominięcie istotnych aspektów. W celu znalezienia odpowiedniej równowagi do prezentowania przykładów wybrałem tzw. obszar niezagospodarowany (ang. *greenfield*)[2].

Przyglądając się projektom w różnej fazie, zauważymy problemy, z jakimi borykają się zespoły, jak również sukcesy, które zespoły odnoszą. Dziedzina Główna, która stanowi sedno przykładów, jest wystarczająco złożona, by można było badać DDD z różnych perspektyw. Nasze Konteksty Ograniczone używają jednego albo kilku innych Kontekstów Ograniczonych, co pozwala badać zagadnienia integracji z DDD. Pomimo to na podstawie trzech przykładowych modeli nie da się zademonstrować wszystkich aspektów projektowania strategicznego, jakie występują powszechnie w środowisku „obszaru brązowego" (ang. *brownfield*) — tzn. takiego, w którym działa wiele tradycyjnych systemów. Staram się, by te mniej atrakcyjne regiony nie zostały zupełnie pominięte, tak jakby były nieistotne. Tam, gdzie jest to wskazane, odbiegam od głównych obszarów przykładów i studiuję obszary, w przypadku których można skorzystać ze wskazówek DDD w dodatkowy sposób, przynoszący większe korzyści.

W dalszej części tekstu zaprezentuję czytelnikom firmę i opowiem trochę o zespołach oraz projektach, nad którymi one pracują.

## Firma SaaSOvation, jej produkty i wykorzystanie DDD

Przykładowa firma nazywa się SaaSOvation. Jak sugeruje nazwa przedsiębiorstwa, SaaSOvation zajmuje się rozwojem aplikacji SaaS (ang. *Software as a Service* — dosł. *oprogramowanie jako usługa*). Firma SaaSOvation zapewnia hosting swoich produktów. Korzystają z nich firmy, które dokonają subskrypcji. Plan biznesowy przedsiębiorstwa obejmuje rozwój dwóch produktów.

[2] Autor ma na myśli zupełnie nowy temat, gdzie projektowanie odbywa się od zera. Jest to tzw. obszar zielony (ang. *greenfield*), w przeciwieństwie do „obszaru brązowego" (ang. *brownfield*), tzn. sytuacji, w której już występują jakieś systemy — *przyp. tłum.*

Flagowy produkt firmy nosi nazwę CollabOvation. To aplikacja wspomagająca współpracę w firmie. Wykorzystano w niej własności najpopularniejszych sieci społecznościowych. Obejmuje to fora, współdzielone kalendarze, blogi, komunikatory, wiki, tablice ogłoszeń, mechanizmy zarządzania dokumentami, powiadamiania i alarmowania, śledzenia aktywności oraz czytników RSS. Wszystkie te narzędzia współpracy grupowej koncentrują się na potrzebach firm korporacyjnych. Pozwalają zwiększyć produktywność w mniejszych projektach, w większych programach oraz na przestrzeni wielu jednostek biznesowych. Współpraca w biznesie jest ważna ze względu na tworzenie i wspomaganie synergicznej atmosfery we współczesnej, zmieniającej się i czasami niepewnej, ale dynamicznej, gospodarce. Wszystko, co może przyczynić się do zwiększenia produktywności, transferu wiedzy, promowania współdzielenia myśli technicznej i asocjacyjnego zarządzania procesem twórczym — tak by można było właściwie wykorzystać jego rezultaty — będzie czynnikiem tworzącym przewagę w równaniu opisującym sukces przedsiębiorstwa. Produkt CollabOvation stanowi dla klientów wartościową propozycję i jest wyzwaniem dla jego deweloperów.

Drugi produkt, pod nazwą ProjectOvation, to kluczowa Dziedzina Główna. Narzędzie służy do zarządzania projektami Agile z wykorzystaniem metodyki Scrum. Jest to framework wspomagający iteracyjne i przyrostowe zarządzanie projektem. ProjectOvation oferuje funkcjonalności zgodne z tradycyjnym modelem zarządzania projektem Scrum — uwzględnia produkt, właściciela produktu, zespół, pozycje rejestru zadań do wykonania, zaplanowane wydania i sprinty. Szacowanie pozycji rejestru jest realizowane za pomocą kalkulatorów wartości biznesowych, wykorzystujących analizy typu koszt-zysk. ProjectOvation zmierza w kierunku zapewnienia jak najlepszego wsparcia dla metodyki Scrum. Zakłada się jednak, że system SaaSOvation przyniesie większe korzyści.

Produkty CollabOvation i ProjectOvation nie rozwijają się według zupełnie oddzielnych ścieżek. Członkowie zespołu SaaSOvation i rada doradców przewidzieli innowację polegającą na dostarczeniu narzędzi współpracy do realizacji zwinnych technik rozwoju oprogramowania. Zgodnie z tym funkcjonalność systemu CollabOvation będzie oferowana jako opcjonalne rozszerzenie produktu ProjectOvation. Nie ulega wątpliwości, że dodatek dostarczający narzędzi współpracy do wspomagania planowania projektu, dyskutowania o cechach funkcjonalnych i historyjkach, prowadzenia dyskusji grupowej i zapewniania wsparcia w zespole oraz pomiędzy zespołami będzie popularny. Firma SaaSOvation przewiduje, że funkcje CollabOvation doda ponad 60 procent subskrybentów systemu ProjectOvation. Trzeba pamiętać, że taka dodatkowa sprzedaż często prowadzi do nowej sprzedaży dodatku jako odrębnego produktu. Gdy po ustanowieniu kanału sprzedaży zespoły rozwoju oprogramowania zobaczą możliwości współpracy w swoim zestawie narzędzi zarządzania projektem, ich entuzjazm wpłynie na pełne przyjęcie kompletnego zestawu narzędzi współpracy w całej firmie. Dzięki temu podejściu do sprzedaży firma SaaSOvation prognozuje dalej, że minimum 35 procent całej sprzedaży produktu ProjectOvation będzie prowadziło do pełnego wdrożenia produktu CollabOvation. Kierownictwo uważa to za konserwatywną ocenę, ale jeśli przewidywania okażą się trafne, będzie to oznaczało duży sukces.

Najpierw skompletowano zespół rozwoju produktu CollabOvation. Znalazło się w nim kilku wprawionych weteranów, ale większość zespołu stanowili średnio zaawansowani deweloperzy. Na pierwszych spotkaniach wskazano metodykę DDD jako proponowane podejście do projektowania i rozwoju. Jeden z dwóch starszych deweloperów używał minimalnego zbioru wzorców DDD w poprzednim projekcie u byłego pracodawcy. Ze

sposobu, w jaki opisał swoje doświadczenia zespołowi, bardziej doświadczony deweloper korzystający z DDD wysnułby wniosek, że w tym projekcie nie stosowano w pełni metod DDD. To, czego używano czasami, określa się terminem DDD-Lite.

DDD-Lite to mechanizmy dobierania podzbioru taktycznych wzorców DDD, ale bez poświęcania pełnej uwagi odkrywaniu, opracowywaniu i poprawianiu Języka Wszechobecnego. Ponadto w technice tej zazwyczaj pomija się używanie Kontekstów Ograniczonych i Mapowania Kontekstu. DDD-Lite koncentruje się w większym stopniu na aspektach technicznych, a najważniejszym celem jest rozwiązywanie problemów technicznych. Może to przynosić korzyści, ale ogólnie rzecz biorąc, nie są one tak duże jak w przypadku stosowania technik modelowania strategicznego. Firma SaaSOvation postanowiła skorzystać z DDD-Lite. W tej sytuacji takie postępowanie szybko doprowadziło do problemów, ponieważ zespół nie rozumiał Poddziedzin ani możliwości wyraźnego określenia Kontekstów Ograniczonych i osiąganego dzięki temu bezpieczeństwa.

Ale mogło być gorzej. Firma SaaSOvation właściwie uniknęła największych pułapek związanych z używaniem DDD-Lite, ze względu na to, że jej dwa główne produkty tworzyły naturalny zbiór Kontekstów Ograniczonych. Dzięki temu modele produktów CollabOvation i ProjectOvation mogły być formalnie posegregowane. Ale to był tylko przypadek. To nie znaczyło, że zespół rozumiał Kontekst Ograniczony. Właśnie dlatego wystąpiły problemy. Albo się czegoś uczymy, albo spotyka nas niepowodzenie.

Na szczęście możemy skorzystać z przeanalizowania niekompletnego użycia DDD w firmie SaaSOvation. Zespół w końcu wyciągnął wnioski ze swoich błędów i w pełniejszy sposób zastosował techniki projektu strategicznego. Na podstawie udoskonaleń wprowadzonych przez zespół CollabOvation dowiemy się także, że utworzony później zespół produktu ProjectOvation skorzystał na retrospektywach wczesnych warunków swojego siostrzanego i partnerskiego projektu. Więcej informacji na ten temat można uzyskać w opisie wzorców **Poddziedzina (2)** i **Kontekst Ograniczony (2)**, jak również **Mapa Kontekstu (3)**.


## Podsumowanie

W niniejszym rozdziale zaprezentowano dość zachęcające wprowadzenie w metodykę DDD. Sądzę, że do tej chwili czytelnik przekonał się, iż zespół może odnieść sukces, jeśli zastosuje zaawansowaną technikę projektowania oprogramowania. Zgadzam się z tym.

Oczywiście nie mam zamiaru nadmiernie upraszczać obrazu. Wdrożenie DDD wymaga sporego wysiłku wielu osób. Gdyby to było łatwe, wszyscy pisaliby

doskonały kod, a przecież dokładnie wiemy, że tak się nie dzieje. Zatem przygotuj się na wysiłek. Warto będzie go podjąć, ponieważ dzięki temu projekt będzie wyrażał dokładnie to, co robi oprogramowanie.

Oto czego nauczyłeś się dotychczas:

- Odkryłeś, co dla Twoich projektów i zespołów może przynieść zastosowanie DDD do zmagania się ze złożonością dziedziny.
- Dowiedziałeś się, jak ocenić, czy Twój projekt kwalifikuje się do zainwestowania w DDD.
- Przeanalizowałeś popularne alternatywy dla DDD i dowiedziałeś się, dlaczego stosowanie tego podejścia często prowadzi do problemów.
- Zapoznałeś się z podstawowymi założeniami DDD i jesteś przygotowany do wykonania pierwszych kroków w swoim projekcie.
- Dowiedziałeś się, jak zaprezentować DDD swojemu kierownictwu, ekspertom dziedziny i technicznym członkom zespołu.
- Teraz jesteś uzbrojony w wiedzę potrzebną do odniesienia sukcesu podczas stawiania czoła wyzwaniom DDD.

Oto czym zajmiemy się dalej. Następne dwa rozdziały dotyczą najważniejszych aspektów projektowania strategicznego. Kolejny rozdział natomiast dotyczy architektury oprogramowania tworzonego z wykorzystaniem metodyki DDD. To naprawdę ważne zagadnienia, z którymi trzeba się zapoznać przed przystąpieniem do lektury dalszych rozdziałów, dotyczących modelowania taktycznego.

# Skorowidz

## D



## E



## F



## G



## H



## I

## J



## K

## L



## Ł



## M

## P

## R



## S

## Ś



## T



## U

## W

## Y



## Z

## Ź



## Ż

**Sprawne budowanie dużych systemów oprogramowania** jest nie lada wyzwaniem, zwłaszcza gdy trzeba spełnić specyficzne wymagania biznesowe. Programowanie dziedzinowe, zwane w skrócie DDD, jest nowatorskim podejściem do projektowania architektury oprogramowania, pozwalającym na szybkie uzyskiwanie pożądanych efektów. Wielu architektów stosuje DDD wyłącznie jako techniczny zbiór narzędzi i nie wykracza poza wykorzystywanie wzorców taktycznych. Tymczasem dopiero pełne wykorzystanie strategicznych wzorców projektowych DDD pozwoli na prawdziwie skuteczne projektowanie skomplikowanych systemów oprogramowania.

**Niniejsza książka** jest przeznaczona dla architektów aplikacji skali korporacyjnej. Zawarto tu wyczerpujący opis zbioru narzędzi DDD i ich stosowania do projektowania różnych systemów, a także w przystępny sposób pokazano aspekty praktycznego wykorzystania nowych technik, takich jak wzorce CQRS czy magazynowanie zdarzeń. Są one stosowane z upodobaniem przez wielu praktyków DDD. Zaprezentowano tu wiele przykładów i cennych wniosków. Jednym słowem, jest to kompletny podręcznik, z którego skorzystają wszyscy deweloperzy oprogramowania, niezależnie od posiadanego doświadczenia.

## W książce przedstawiono następujące zagadnienia:

- wprowadzenie do DDD i główne zasady tego podejścia do projektowania
- zastosowanie DDD w różnych architekturach, włącznie z architekturą sześciokątną, SOA, REST, CQRS, sterowaną zdarzeniami oraz Data Fabric (Grid)
- zasady projektowania z wykorzystaniem encji i obiektów wartości
- praktyczne zastosowania takich technik DDD jak zdarzenia dziedziny, moduły, agregaty
- zasady implementacji integracji modelu z wykorzystaniem mapowania kontekstu oraz dziedziny głównej z kontekstami ograniczonymi
- techniki projektowania repozytoriów dla rozwiązań ORM, NoSQL i wielu innych

**Vaughn Vernon** — projektant odpowiedzialny za rozwój architektury oprogramowania. Uznany lider nowatorskiego podejścia do upraszczania projektu i implementacji oprogramowania. Zasady programowania dziedzinowego stosuje w praktyce od lat dziewięćdziesiątych, tworząc modele oprogramowania dla takich branż jak zarządzanie przestrzenią powietrzną, ochrona środowiska, ubezpieczenia, ochrona zdrowia czy telekomunikacja. Jest uznanym autorytetem w dziedzinie DDD — jego wykłady cieszą się wielką popularnością w wielu krajach.

## Z DDD zaimplementujesz wszystko, co zechcesz!



Helion

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-2547-0

9 788328 325470

Informatyka w najlepszym wydaniu

cena: 99,00 zł