# Domain Driven Design krok po kroku

Część VI: Modeling Whirlpool – iteracyjny proces modelowania

<<LEAD>>
Modeling Whirlpool jest oficjalną metodyką DDD służącą iteracyjnemu odkrywaniu modelu domenowego. Metodyka jest zorientowana na tworzenie precyzyjnego słownictwa i precyzyjnych reguł domenowych rozumianych tak samo przez Eksperta Domenowego jak i Modelarza.
<</LEAD>>

## *Plan serii*

Niniejszy tekst jest kolejnym artykułem z serii mającej na celu szczegółowe przedstawienie kompletnego zestawu technik modelowania oraz nakreślenie kompletnej architektury aplikacji wspierającej DDD.

- Część 1: Podstawowe Building Blocks DDD;
- Część 2: Zaawansowane modelowanie DDD – techniki strategiczne: konteksty i architektura zdarzeniowa;
- Część 3: Szczegóły implementacji aplikacji wykorzystującej DDD na platformie Java – Spring Framework;
- Część 4: Skalowalne systemy w kontekście DDD - architektura CqRS;
- Część 5: Kompleksowe testowanie aplikacji opartej o DDD;
- Część 6: Modeling Whirlpool

## *Wstęp*

Celem artykułu jest przedstawienie oficjalnego procesu modelowania DDD: Modeling Whirlpool oraz narzędzi umożliwiających tworzenie wykonywalnej dokumentacji wymagań.

Modeling Whirlpool promuje główne założenie DDD: wypracowanie Ubiquitous Language, którym posługują się wszyscy uczestnicy projektu, co skutkuje

znacznie szybszym cyklem dostrajania modelu oraz pozwala na uświadomienie kosztów zmian oraz zaciąganych długów technicznych.

Natomiast proponowane narzędzia pozwalają na utrzymanie stałej transparencji realizacji wymagań dzięki możliwości uruchomienia wypracowanych scenariuszy jako wykonywalnej specyfikacji, która jest jednocześnie zrozumiała dla nietechnicznych członków zespołu.

## *Projekt referencyjny*

Wszystkich tych Czytelników, którzy już teraz chcieliby zapoznać się z kolejnymi zagadnieniami naszej serii, zapraszam do odwiedzenia strony projektu „DDD&CqRS Laven", której adres znajduje się w ramce „W sieci".

Dostępne wersje:

- Java – Spring
- Java – EJB 3.1
- .NET - C#

## *Inkrementacje to nie Iteracje*

Modeling Whirlpool wspiera podejście iteracyjne, w którym to dostarczamy z iteracji na iterację coraz bardziej uszczegółowiony model oraz implementację wymagań.

Podejście iteracyjne jest często mylone z podejście inkrementacyjnym, w którym po prostu dzielimy większy problem na mniejsze i dostarczamy kolejne porcje rozwiązania. Całość możemy wówczas zweryfikować dopiero gdy zbierzemy wszystkie porcje.

Odnosząc się do pierwszej i drugiej części naszej serii przypomnę, że Building Blocks DDD są zorientowane na podejście iteracyjne. Dzięki np. hermetyzacji agregatów, stosowaniu Value Objects jako projekcji oraz podejściu funkcyjnemu w postaci Polityk, nasz model jest otwarty na proces **Knowledge Crunching** – czyli iteracyjnego uszczegóławiania implementacji poszczególnych Building Blocks.

## *Role w zwinnym procesie modelowania DDD*

W procesie Modeling Whirlpool uczestniczą: Ekspert Domenowy, Modelarz (projektant) i opcjonalnie Animator (Facilitator).

**Ekspert** powinien posiadać głęboką wiedzę o swej domenie – przykładowo będzie nim najstarszy księgowy w firmie klienta. Generalnie jest to osoba, do której zwracają się inny pracownicy ze strony biznesu w razie poważnych problemów.

**Modelarz** powinien mieć pewne doświadczenie programistyczne, ponieważ zgodnie z założeniem model w DDD ma dosłowne odzwierciedlenie w kodzie warstwy domenowej. Jak pamiętamy, warstwa ta nie zawiera szczegółów technicznych związanych z frameworkiem czy platformą, tak więc Modelarz nie musi być ekspertem technicznym.

Należy podkreślić, że nie każdy programista może grać rolę modelarza. W tej roli preferowana jest raczej inteligencja werbalna niż algorytmiczna oraz wymagana jest oczywiście biegłość w posługiwaniu się technikami modelowania DDD – zarówno wzorcami taktycznymi jak i przede wszystkim strategicznymi.

**Animator** może być przydatną osobą w momencie pojawienia się impasu. Animator potrafi zadać trafne pytania dzięki ogólnej wiedzy technicznej oraz domenowej.

Zadaniem animatora jest nadzorowanie sesji aby nie poddała się „dryfowaniu". Dyskusje trwające więcej niż 3h nie mają zwykle sensu, tak więc wówczas Animator powinien przerwać taką sesję i zastosować techniki przełamania impasu.

## *Nieustanny Feedback*

Dzięki temu, że sesja modelowania odbywa się w jednoczesnej obecności wszystkich uczestników unikamy zjawiska przesyłania maili z nieczytelną dokumentacją oraz straty czasu na „odkodowanie" intencji.

## *Do czego służy model*

Jak już wielokrotnie wspomnieliśmy, model służy jako narzędzie

komunikowania pojęć i reguł biznesowych. W DDD model jest dokładnie oddany w kodzie, co zapewnia, że system działa tak jak rozumie to Ekspert Domenowy.

## Ubiquitous Language – co to znaczy w praktyce

Przedstawiony na listingu Serwis Domenowy jest odzwierciedlonym w kodzie modelem operacji wystawienia faktury.

Co dokładnie mówi nam ten model:

- fakturę wystawiamy na podstawie całego zamówienia
- nie ma możliwości wystawienia faktury na poszczególne pozycje
- nie ma możliwości wystawienia faktury zbiorczej na kilka zamówień
- możemy obliczać podatki dla dowolnego kraju, jest to domknięcie operacji issuance, niezależne od adresu klienta na zamówieniu

<<LISTING lang=java>>
@DomainService

public class BookKeeper {

    public Invoice issuance(Order order, TaxPolicy taxPolicy){
      //... }

}

<</LISTING>>
Listing 1. Serwis Domenowy modelujący operację wystawienia faktury.

## Skutek uboczny: rozumienie złożoności i kosztu zmian

Dzięki tak dokładnemu modelowaniu widzimy ograniczenia modelu. Nie możemy fakturować kilku zamówień ani poszczególnych pozycji. Zmiana reguł domenowych nie polega jedynie na wysłaniu maila o treści typu „dodajcie tylko jeden checkbox na ekranie”. Ekspert domenowy wykonuje zmianę na modelu. Daje to możliwość wychwycenia błędów logicznych już na poziomie modelu.

Poza tym zasięg zmiany oraz jej „dotkliwość” są w niemal namacalny sposób odczuwalne przez biznes. Wreszcie mamy odpowiedź na pytanie „dlaczego

zmiana kosztuje tak dużo?”.

## Skutek uboczny: świadome zaciąganie długu technicznego i ograniczania punktów swobody modelu

Wszelkie uproszczenia są w modelu DDD wprowadzane świadomie z całą konsekwencją ich skutków. Dokładnie widzimy, w których miejscach model posiada punkty otwierające go na zmiany, a w których miejscach jest on zamknięty na rozbudowę.

Oczywiście nie możemy zaprojektować całego systemu w sposób otwarty na rozbudowę, ale mamy obraz potencjału oraz świadomość, tego jak nasze decyzje modelowe wpływają na dług techniczny.

## *Wir modelowania*

### *1. Symptomy – po czym poznać, że należy zacząć modelować*

Oczywistym powodem do rozpoczęcia modelowania jest podjęcia prac nad nową domeną – np. kolejnym Bounded Context.

Typowym symptomem sugerującym konieczność pochylenia się nad modelem są problemy w komunikacji. Przykładowo zespół developerski zaczyna używać pojęć niezrozumiałych dla Eksperta Domenowego – czystych fabrykatów wprowadzonych z powodu ograniczeń technicznych aktualnego modelu. Są to słowa typu „FakeOrder”, „TemporaryItem”.

Kiedy model staje zbyt złożony i wymaga „hackowania” błędnych koncepcji zaczynamy obserwować „brodzenie” - czyli wyraźne spowolnienie prac, mierzone np. poprzez Scrum Velocity.

Starsi programiści z wyrobioną intuicją zauważają niepokojący symptom: złożoność rozwiązania zdaje się być niewspółmiernie większa niż odczuwana intuicyjnie złożoność samego problemu. Mamy wówczas do czynienia z eskalacją Accidental Complexity, która przerasta Essential Complexity [zob. Ramka „W sieci” – teoria złożoności].

## 2. Burza mózgów

Burzę mózgów rozpoczynamy od przedstawienia referencyjnych scenariuszy. Scenariusze te mogą pochodzić z nowej domeny lub być kłopotliwymi przypadkami, które doprowadziły do zainicjowania sesji modelowania. Podczas fazy Burzy Mózgów zbieramy alternatywne projekty domeny pamiętając o wszystkich zasadach brainstormingu. Przede wszystkim nie oceniamy pomysłów, choćby wydawały się pozbawione elementarnego sensu – pamiętajmy, że procesy umysłowe potrzebują czasem stymulacji również tego typu.

Osoby biorące udział w burzy mózgów będą charakteryzować się różnymi strategiami kognitywnymi. Niektóre osoby mają tendencję do drążenia szczegółowych przykładów. Nie należy wówczas przerywać tego typu dyskusji, ponieważ pewne strategie kognitywne pomimo tego, że polegają na wnikaniu w szczegóły i orientacji na różnice, to w efekcie są w stanie doprowadzić do ogólnych modeli.

## 3. Scenariusz

W tej fazie Ekspert Domenowy opowiada Story. Unikanie stylu pasywnego i długich, złożonych zdań. Warto przyjąć styl zorientowany na opis sekwencji wydarzeń biznesowych - niekoniecznie musi być to kompletny proces.

Skupiamy się na jednej ścieżce i jej istotnych krokach. Wybieramy krytyczne dla procesu kroki – wyznacza jest granica Core Domain.

Pozostali uczestnicy sesji muszą w 100% zrozumieć Story (wszystkie pojęcia) oraz powód dlaczego jest istotna.

## 4. Model

W fazie modelowania warto skupić się na zmianach stanu modelu w każdym kroku scenariusza. Z tych zmian wyłaniają się niezmienniki naszych Agregatów (zobacz część II serii).

Na bieżąco sprawdzamy, czy język jakim posługuje się modelarz wciąż brzmi naturalnie dla eksperta domenowego – czy nie wprowadzamy własnego ani

dwuznaczności.

Model powinien być raczej "thing oriented" niż "process oriented" - procesy częściej się zmieniają. Warto posłużyć się techniką 4 poziomów modelu (zobacz część II serii).

Na bieżąco sprawdzamy jak „pracuje” model, czyli czy da się przeprowadzić przez niego określony w poprzedniej fazie scenariusz odnośny. Technicznie rzecz biorąc sprawdzamy „wywołania” metod naszych Building Blocks walidując czy aby na pewno wszystkie informacje wejściowe i wyjściowe są dostępne oraz czy spełniamy niezmienniki operacji.

## 5. Wyzwanie

Jak będzie pracował model, gdy spróbujemy „przeprowadzić przez niego” inny ważny lub trudny scenariusz? Czy wywołania operacji Building Block można ułożyć tak, aby realizowały inny sensowny scenariusz?

Mamy ty na myśli zarówno kolejny scenariusz z puli wymagań jak i inny wyimaginowany (ale sensowny) scenariusz – dzięki temu sprawdzimy punkty swobody modelu i wykryjemy ew. długi techniczne.

W razie problemu powrót do fazy Modelowania.

Warto przyjąć motto: **„używanie zbyt prostych modeli do zbyt złożonych problemów prowadzi do wielkich komplikacji”**.

## 6. Próba

W fazie próby chcemy stworzyć kod źródłowy "chodzącego szkieletu" (walking skeleton). Skupiamy się na klasach i nagłówkach metod nieimplementując ich treści. Gdyby okazało się, że model nie jest trafnym, to wówczas implementacji okazałaby się stratą czasu. Implementacja szczegółów modelu będzie zadaniem na bieżącą iterację.

Podczas tworzenia szkieletu mogą pojawić się niespójności, których nie wykryliśmy w fazie Wyzwania – po prostu kompilator nie wybacza (chyba, że używany języka dynamicznego bez silnego typowania:)

W fazie tej pamiętajmy, że naszym celem jest skupienie się na tym co "należy udowodnić", nie na "bezpiecznych" zadaniach, tak więc eliminujemy proste zadania typu operacji CRUD.

### *Strategia: Wykonywalne scenariusze akceptacyjne*

Zebrane podczas wiru modelowania scenariusze możemy wykorzystać jaką wykonywalną dokumentację. W tym celu wprowadzimy: formalizację ich struktury oraz narzędzia pozwalające na wykonanie tekstu scenariusza wobec tworzonego systemu.

W efekcie w procesie produkcji oprogramowania pojawi się dokumentacja, wymagań, która jest zawsze aktualna oraz transparentna – zawsze mamy w wgląd w postęp prac – widzimy, które scenariusze zostały zaimplementowane i działają zgodnie ze specyfikacją.

<<RAMKA>>

### Taktyka: Struktura scenariuszy

[Nazwa Story]

As a [Rola w systemie]

In order to [korzyść, cel]

I want to [historyjka opisująca interakcję z systemem]

Background:

[Stan systemu przed każdym scenariuszem]

Scenario: [Tytuł pierwszego scenariusza]

Given [Stan systemu] and [Stan systemu] and …

When [aktywność użytkonika]

Then [Stan systemu] and [Stan systemu] and …

(potencjalnie kolejna seria Given When Then)

Scenario: [Tytuł kolejnego istotnego scenariusza]

(struktura taka sama jak powyżej)

<</RAMKA>>

### Szablon User Story

Szablon User Story z ramki „szablon scenariuszy” posiada charakterystyczną nazwę oraz wstępną narrację, która opisuje cel, jaki chce osiągnąć użytkownik grający daną rolę w systemie. Sformułowanie celu jest bardzo ważne, ponieważ kolejne scenariusze będą do niego wyraźnie dążyły – jednak każdy scenariusz w inny sposób.

Na każdy User Story składa się wiele scenariuszy akceptacyjnych, z których każdy na swój specyficzny sposób dąży do określonego w narracji celu.

Scenariusz posiada tytuł, który powinien go w jasny sposób odróżniać od innych scenariuszy, wyjaśniając co takiego szczególnego jest w tym scenariuszu.

Scenariusz składa się z sekcji Given, When, Then. Złożone scenariusze mogą być rozbudowywane zarówno „w prawo” - dodając słowo kluczowe And, jak również „w dół” dodając kolejne sekcje  Given, When, Then.

Scenariusze dodajemy wraz z kolejnymi cyklami Modeling Whirlpool, co przekłada się na iteracyjny (nie inkrementacyjny) proces uszczegóławiania wymagań.

### Przykład

<<LISTING lang=story>>

```
Customer orders products that are in stock

Narrative:
In order to buy interesting products
As a customer
```

```
I want to browse the available products, pick the ones that I like and order them

Background:
All products are in stock

Scenario: standard client creates standard order
Given Products are available
When I add any product to basket
Then Basket should contain 1 item
When I checkout
Then I should be able to submit order with 1 product
When I submit the order
Then That order should be confirmed

Scenario: indebted client creates standard order
…

Scenario: indebted client creates extraordinary order
...
```

<</LISTING>>

Listing 1. Przykładowe scenariusze. Plik src/test/resources/productsOrdering.story

### Dobre praktyki

Stworzenie dobrze zdefiniowanych scenariuszy na odpowiednim poziomie ziarnistości jest sztuką samą w sobie i wymaga zarówno doświadczenia jak i dopasowania do natury projektu nad którym pracujemy.

Możemy jednak wyszczególnić kilka dobrych zasad, które sprawdzają się niemal zawsze:

- Kroki story są zorientowane na „flow” a nie na interakcję z GUI. Przykładowo krok „`I add any product to basket`” może przekładać się na interakcję z kilkoma kontrolkami graficznymi (wpisanie ilości, kliknięcie dodaj i kliknięcie zatwierdź), ale nie jest to istotne z punktu widzenia eksperta domenowego. Aczkolwiek może być krytyczne dla eksperta od ergonomii interfejsu użytkownika czy architekta informacji.
- Scenariusze dążą do celu zdefiniowanego w narracji, jednak każdy różni się czymś szczególnym. Każdy scenariusz można rozumieć jako osobny „slalom” do celu.
- Jedną z praktyk (która może mieć zastosowanie w niektórych sytuacjach, szczególnie w systemach, które posiadają już bogatą funkcjonalność) jest specyfikowanie Given i Then przez Eksperta Domenowego, natomiast When przez modelarza, który np. ma wiedzę o już istniejących usługach, które mogłyby być wykorzystane do realizacji story.
- Scenariusze w Modeling Whirlpool powinny być zorientowane wokół obiektów z warstwy domenowej – w odróżnieniu od scenariuszy z Behavior Driven Development, które są zorientowane raczej na interakcję z systemem.

### Taktyki: Dwuwarstwowa architektura wykonywalnych scenariuszy

Scenariusze w postaci plików tekstowych (ew. formacie wiki, html, itd. w zależności od narzędzia) mogą zostać wykonane. Oczywiście nie mamy tutaj do czynienia z magią. Potrzebujemy kodu, który wykona kroki wobec implementowanego systemu. Przykład takiego kodu znajduje się na listingu 2. Będziemy potrzebować również narzędzia, które dokona mapowania scenariuszy z plików tekstowych na tenże kod. W projekcie referencyjnym posłużyliśmy się narzędziem Jbehave, który dokonuje mapowanie poprzez adnotacje.

<<LISTING lang=java>>

```
@Steps
public class ProductsOrderingSteps {
```

```
    @Inject
    private ProductsListAgent productsList;

    @Inject
    private OrderConfirmationAgent orderConfirmation;

    @Given("Products are available")
    public void productsAreAvailable() {
        // assume there are products
        assertTrue(productsList.productsExist());
    }

    @When("I add any product to basket")
    public void addAnyProductToBasket() {
        productsList.addAnyProductToBasket();
    }

    @Then("Basket should have $number items")
    @Alias("Basket should contain $number item")
    public void shouldBasketHaveItems(int itemsCount) {
        assertEquals(itemsCount,
productsList.getBasketItemsCount());
    }

    @When("I checkout")
    public void checkout() {
        productsList.checkout();
    }
```

<</LISTING>>
Listing 2. Fragment kodu kroków scenariuszy z klasy pl.com.bottega.acceptance.erp.steps.ProductsOrderingSteps

Ważne jest tutaj uchwycenie dwóch poziomów abstrakcji:

- Na poziomie scenariuszy mamy „flow” użytkownika, który abstrahuje od szczegółów interakcji z systemem. Jest to warstwa Flow.
- Natomiast na poziomie kroków w kodzie wykonujemy interakcje z systemem. W naszym przykładzie każdy krok to jednak interakcja z systemem, ale nie musi tak być. Jest to warstwa automatyzacji scenariusza.

Zwróćmy uwagę na fakt, że interakcje z systemem odbywają się abstrakcyjnego agenta, np. ProductListAgent, OrderConfirmationAgent, który jest interfejsem do naszego systemu i abstrahuje od technologii komunikacji. Może być zarówno Agent, który „klika” w komponenty graficzne na stronie internetowej przy pomocy Jbehave jak i Agent, który komunikuje się zdalnie z usługami serwera (np. EJB, Spring Remoting, Web Service, Rest, …).

Szczegóły implementacji Agentów z wykorzystaniem Selenium i Spring Remoting można znaleźć w projekcie referencyjnym – adres w ramce „W sieci”.

Jedną z praktyk, która może zastosowanie w niektórych projektach jest podejście w którym:

- krok When jest wykonywany przez GUI
- kroki Given i Then są wykonywane przez API servera (ew backdoors)

Dzięki takiemu podejściu unikamy kosztów związanych z utrzymaniem automatyzacji testów GUI.

### *Strategia: Wykonywalne specyfikacje z przykładami*

Wprowadzenie dwóch warstw wykonywalnych specyfikacji jest często przełomem w projekcie. Górna warstwa jest zorientowana na „flow” użytkownika, a dolna zajmuje się szczegółami automatyzacji interakcji z GUI. Dzięki temu unikamy powstania „skryptów klikania” i związanych z nimi

kosztów utrzymania.

Ale czasem zorientowanie na „flow” nie jest odpowiednim podejściem do specyfikowania wymagań. Zależy to od natury problemu a często nawet od sposobu rozumowania Eksperta Domenowego.

Możemy wprowadzić trzecią warstwę wykonywalnej specyfikacji: technikę Specification by Example. Omówienie tej techniki wykracza daleko poza zakres tego artykułu. Przedstawimy tutaj jedynie przykład, a zainteresowanych odsyłamy do doskonałej książki: „Specification by Example: How Successful Teams Deliver the Right Software” autorstwa Gojko Adzic.

W podejściu tym nie orientujemy się nawet na „flow” a na pewne reguły oraz podajemy ich przykłady. Jest to kolejna warstwa, ponieważ metody reguł wywołują metody kroków scenariuszy.

<<RAMKA>>

## Problemy, strategie , taktyki i techniki Testowania – kontynuacja z części V

Problemy:

- Eksplozja kombinatoryczna przypadków – w górnych warstwa multiplikują się możliwe stany obiektów domenowych
- Nieaktualna dokumentacja (nikt jej nie czyta ani anie aktualizuje)
- Problem z komunikacją - brak zrozumienia celów biznesowych, biznes nie rozumie systemu
- Kosztowne w utrzymaniu skrypty do „wyklikania”
- Rola testów: jakość z punktu widzenia zespołu czy jakość z punktu widzenia klienta.

Strategie:

- Perfekcja domeny, ogólna zgodność wymagań
- Twórz wykonywalną dokumentację

- Skupiaj dokumentację wokół celów biznesowych i flow użytkownika
- Operuj słownictwem domenowym
- Orientuj się na Flow a nie na skypt UI
- Orientuj się na Specyfikację a nie Flow
- Ciągła refaktoryzacja

Taktyki:

- Dwuwarstwowe Historyjki akceptacyjne
- Trójwarstwowe Wykonywalne Specyfikacje
- Struktura Given-When-Then: W proponuje dev, GT definiuje ekspert
- Testowanie przed refaktoryzacją (odpowiednie testy)

Techniki:

- Testowanie niezmienników Agregatów
- Agenty abstrahujące od protokołu komunikacyjnego
- Struktura Given-When-Then: When sprawdzaj przez warstwę UI, Given i Then przez warstwę API
- Niektóre specyfikacje testuj jednostkowo w warstwie domeny
- Techniki refaktoryzacji

<</RAMKA>>

<<RYSUNEK>>
<<graphic place=GRAFIKA file_name=warstwy_spec.tif/>>
Rysunek 1. Warstwy wykonywalnych specyfikacji
<</RYSUNEK>>

### *Proces Agile: scenariusze behawioralne i wykonywalne specyfikacje*

Na wyższym poziomie tworzymy:

- wykonywalne scenariusze akceptacyjne sprawdzające zgodność zachowania systemu z User Story na poziomie flow użytkownika,
- wykonywalne Specification by Example sprawdzające realizację celów

biznesowych stojących ponad flow użytkownika.

W przypadku tych technik nie mówimy już o testach, a o wykonywalnej dokumentacji wymagań.

Kolejną część naszej serii poświęcimy omówieniu:

- kompletnego procesu modelowania DDD „Modeling Whirlpool”,
- procesu wytwórczego Behavior Driven Development (zwanego Agile drugiej generacji)
- narzędzi wspierający BDD i Specification by Example: Jbehave i Selenium

<<W_SIECI>>

- oficjalna strona DDD http://domaindrivendesign.org
- wstępny artykuł poświęcony DDD http://bottega.com.pl/pdf/materialy/sdj-ddd.pdf
- przykładowy projekt: http://code.google.com/p/ddd-cqrs-sample/
- Modeling Whirlpool http://domainlanguage.com/ddd/whirlpool/
- Jbehave http://jbehave.org/
- Behavior Driven Development http://dannorth.net/introducing-bdd/
- Teoria złożoności http://en.wikipedia.org/wiki/Accidental_complexity http://en.wikipedia.org/wiki/Essential_complexity

<</W_SIECI>>

<<O_AUTORZE posx=9;0l posy=b fit=W grow=H>>

Sławomir Sobótka

Programujący architekt aplikacji specjalizujący się w technologiach Java i efektywnym wykorzystaniu zdobyczy inżynierii oprogramowania. Trener i konsultant w firmie Bottega IT Solutions. Entuzjasta Software Craftsmanship.

Do jego zainteresowań należy szeroko pojęta inżynieria oprogramowania: architektury wysokowydajnych systemów webowych (w szczególności CqRS), modelowanie (w szczególności DDD), wzorce, zwinne procesy wytwórcze. Hobbystycznie interesuje się psychologią i kognitywistyką.

W wolnych chwilach działa w community jako: prezes Stowarzyszenia Software Engineering Professionals Polska (http://ssepp.pl), lider lubelskiego Java User Group, publicysta w prasie branżowej i blogger (http://art-of-software.blogspot.com).

Kontakt z autorem: slawomir.sobotka@bottega.com.pl
<</O_AUTORZE>>