*Autor*

# Receptury projektowe – niezbędnik początkującego architekta

## Część III: 4C - Zwinne podejście do odkrywania i dokumentowania architektury

W jaki sposób dokumentować architekturę systemu? Z jednej strony tak, aby zawrzeć wszystkie potrzebne informacje, z drugiej zaś, aby nie przeładować dokumentacji szczegółami, które czynią ją bezużyteczną. W artykule zostanie przedstawione jedno z podejść do dokumentowania architektury na 4 poziomach: Context, Containers, Components, Classes.

### O serii „Receptury projektowe"

Intencją serii „Receptury projektowe" jest dostarczenie usystematyzowanej wiedzy bazowej początkującym projektantom i architektom. Przez projektanta lub architekta rozumiem tutaj rolę pełnioną w projekcie. Rolę taką może grać np. starszy programista lub lider techniczny, gdyż bycie architektem to raczej stan umysłu niż formalne stanowisko w organizacji.

Wychodzę z założenia, że jedną z najważniejszych cech projektanta/architekta jest umiejętność klasyfikowania problemów oraz dobieranie klasy rozwiązania do klasy problemu. Dlatego artykuły będą skupiały się na rzetelnej wiedzy bazowej i sprawdzonych recepturach pozwalających na radzenie sobie z wyzwaniami niezależnymi od konkretnej technologii, wraz z pełną świadomością konsekwencji płynących z podejmowanych decyzji projektowych.

## OBRAZ WART...

Podobno dobry obraz jest jest wart tysiąca słów. Jednak słaby diagram architektoniczny jest wymienialny co najwyżej na tysiąc inwektyw rzucanych przez odbiorców w kierunku jego twórcy.

Typowe problemy z diagramami:

- są zbyt ogólne – brak w nich wartościowej informacji
- są zbyt szczegółowe – szum informacyjny, wymagają „dekodowania"
- dokumentują oczywiste fakty (chociaż być może były to przełomowe odkrycia dla ich twórców)
- ukrywają esencjonalną złożoność (zwykle w prostokątach z opisem „Bus")
- są jedynie wizualizacją stosu technologii i frameworków
- są sumą wielu warstw informacji
- wyglądają jak słowniki pojęć (ew. pogrupowane przerywanymi liniami lub prostokątami)
- wyglądają jak mapy z gier platformowych - osobnym kolorem zaznacza się na nich „podróż bohatera" (np. szczególny Use Case)
- wyglądają jak mapy korytarzy powietrznych nad dużym lotniskiem (więcej kolorów i przerywanych kresek niż w przypadku powyżej)
- są wygenerowane przez automatyczne narzędzia – niestety ich odczyt wymaga posiadaniu syndromu Aspergera (syndrom ten mieści się w spektrum Autyzmu)

## PROBLEM Z METAFORĄ

Nasza branża przez wiele lat próbowała posiłkować się metaforami z branży budowlanej. Architekturę oprogramowania przekładano na architekturę budynków. Paralela ta była bardzo popularna, dopóki IT nie zaczęło się bliżej przyglądać budownictwu, po czym zrozumiano kuriozalną nieadekwatność tego przełożenia.

Od kilku lat nieśmiało promuje się metaforę ogrodnictwa. Jednak słowo *ogrodnik* nie ma w sobie takiego ładunku splendoru jak słowo *architekt*, dlatego metafora ta jest skazana na uschnięcie niczym kwiatek na pustyni.

Ostatnio jednak metafory z branży budownictwa powracają – jednak w innej skali. Jeżeli spojrzeć na rozległy system jak na miasto, to możemy zaobserwować szereg zjawisk:

- kontekst – każde miasto jest inne z uwagi na uwarunkowania (rzeźba terenu, akweny wodne)
- komponenty – wyraźnie wydzielone strefy o konkretnym przeznaczeniu
- hierarchiczność – pewne kluczowe komponenty (np. o znaczeniu komunikacyjnym) skupiają wokół siebie inne komponenty
- rozbudowa – jeżeli system prosperuje, to jest rozbudowywany
- przebudowa – pewne tymczasowe rozwiązania są przebudowywane – np. przeskalowanie tras komunikacyjnych, w miejscach gdzie wcześniej nie spodziewano się ruchu
- wyłaniające się wzorce – podczas rozbudowy, prowadzącej do przebudowy, zauważamy wzorce wyższego rzędu, które aplikujemy dopiero teraz; wcześniej posiadaliśmy zbyt mało wiedzy o kontekście, aby podjąć decyzję o wprowadzeniu wielkoskalowego wzorca
- ograniczenia – pewnych decyzji nie da się (tanio) zmienić

Tak więc podchodząc do projektu większego systemu, będziemy traktować jako rozrastające się miasto, w którym z czasem wyłonią się struktury i pojawi się miejsce na „nałożenie" wzorców architektonicznych. Dodam od razu, że być może pewne elementy systemu (moduły/komponenty) są być może od razu dobrze określone. Na tym poziomie szczegółowości (metafora pojedynczego budynku) być może faktycznie w pewnych sytuacjach możemy z góry przyjąć pewną architekturę opartą na sprawdzonych wzorcach.

Dlatego wyraźnie różnimy tutaj dwie skale: architekturę systemu i architekturę aplikacji. Ta pierwsza będzie służyć do wyłonienia się struktury unikatowego systemu (złożonego ze zintegrowanych podsystemów) – systemu, który powstaje w unikatowym kontekście. Jej metaforą będzie miasto. Ta druga natomiast będzie strukturą pojedynczego komponentu na poziomie jego kodu źródłowego. Jej metaforą w szczególnych przypadkach może być budynek.

## PODEJŚCIE 4C: CONTEXT, CONTAINERS, COMPONENTS, CLASSES

W podejściu 4C zakładamy, że niemożliwe jest przedstawienie architektury przy pomocy diagramów o jednym poziomie „głębokości". Dlatego two-

rzymy je na co najmniej 4 poziomach. Na każdym poziomie inaczej podchodzimy do aspektów:

- cel – w jakim celu powstaje ten diagram, co powinien przedstawiać
- struktura – z jakich building blocks składa się diagram
- motywacja – co chcemy przekazać odbiorcom dokumentu oraz jakie odkrycia możemy poczynić podczas jego tworzenia
- odbiorcy – kim są odbiorcy i czego mogą potrzebować na danym poziomie szczegółowości

# POZIOM 1: CONTEXT – „I CO JA ROBIĘ TU?"

Pierwszy poziom diagramów w podejściu 4C pozwala nabrać ogólnego wyobrażenia o systemie, jego przeznaczeniu i kontekście, w jakim system będzie pracował. Zwykle jest to po prostu jeden diagram.

## Cel

Tworząc ten diagram, powinniśmy odpowiedzieć sobie na pytania:

- Czym jest system
- Kto/Co go używa – jakie są główne role?
- Jak wpasowuje się on w istniejące środowisko IT

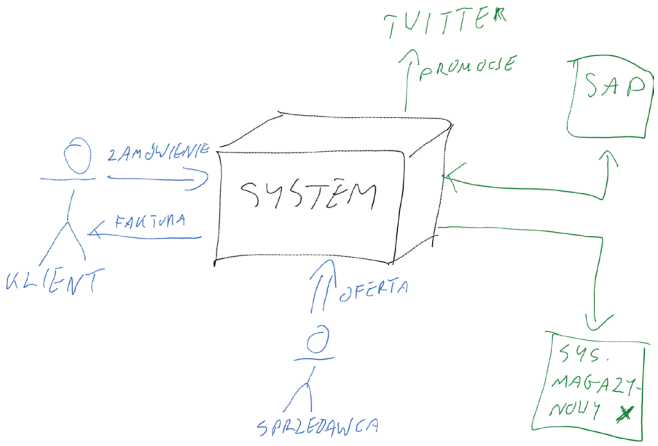
## Struktura

Pomijamy szczegóły, skupiając się na rolach użytkowników i innych systemach. Zacznijmy od naszego systemu zlokalizowanego w centrum, po czym dodajemy kluczowe role i inne systemy wchodzące w interakcje. Skupiamy się na celu tych interakcji – jakie potrzeby realizuje system oraz jakie potrzeby system posiada.

Pomocne na tym poziomie może być również zilustrowanie głównych procesów biznesowych (na ogólnym poziomie), które pozwolą zobaczyć, jak potrzeby aktorów są realizowane przez system i jego otoczenie.

## Motywacja i Odbiorcy

Diagram na takim poziomie ogólności jest polem do dyskusji pomiędzy technicznymi i nietechnicznymi uczestnikami projektu. Diagram dobrze pokazuje to, co będzie dodane do istniejących rozwiązań IT.




*Rysunek 1. Diagram ilustrujący kontekst, w jakim będzie pracował system*

# POZIOM 2: CONTAINERS – „PUDEŁKA" DLA KOMPONENTÓW

Na drugim poziomie szczegółowości zajmujemy się kontenerami, w jakich „rezyduje" system. Mniej złożone systemy mogą rezydować w jednym kontenerze (np. serwerze aplikacyjnym, urządzeniu) – wówczas możemy pominąć ten poziom i przejść do trzeciego.

Natomiast bardziej złożone systemy rezydują w klastrach serwerów aplikacyjnych, połączonych z urządzeniami zewnętrznymi (nadajniki, sterowniki maszyn), zintegrowanych z innymi podsystemami, serwerami mediów, bazy danych (jakiego rodzaju), czasem końcówki klienckie potrzebują własnych maszyn, gdzie integrują się istniejącym oprogramowaniem.

## Cel

Główne pytania, na jakie powinny odpowiadać diagramy z poziomu Containers:

- Wysokopoziomowe decyzje techniczne (np. podział systemu, dobór rozwiązań i technologii, architektura failover)
- Generalna komunikacja pomiędzy kontenerami
- Ewentualne wzbogacenie o informacje z poziomu kontekstu (role i systemy trzecie)
- Informacja dla developerów: gdzie należy „przyłożyć rękę", aby stworzyć rozwiązanie
- Wyznaczenie technicznej granicy systemu

## Struktura

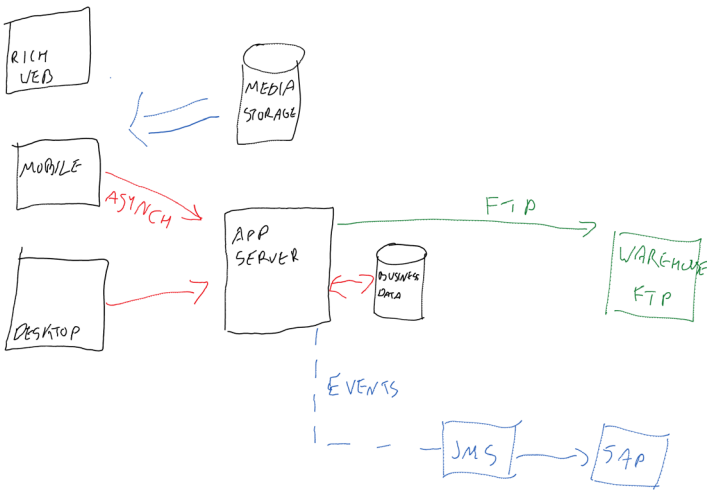
Najbardziej odpowiedni jest diagram bloków, który może zostać wzbogacony o określenie granicy systemu i ew. stref bezpieczeństwa. Warto umieścić: Logiczną nazwę kontenera oraz jego technologię (np. web server Tomcat 7), listę wysokopoziomowych odpowiedzialności (jeżeli nie jest oczywista) oraz wysokopoziomową mapę komunikacji. Dokumentując komunikację, warto skupić się na celu interakcji (np. zapisuje zdjęcia) oraz zaznaczyć informacje co do protokołu (synchroniczny/asynchroniczny, REST, FTP).

## Motywacja i Odbiorcy

Diagramy tego typu pozwalają na komunikację zespołów developerskich (zarówno pracujących nad systemem oraz przygotowujących interfejsy do integracji) oraz Działu Operations.

Tworzymy je, aby ilustrować wysokopoziomowe decyzje techniczne, relacje pomiędzy kontenerami, dyskutować kwestie bezpieczeństwa, wydajności i skalowania.

Przede wszystkim jest to „rusztowanie", na którym będziemy rozpinać komponenty.




*Rysunek 2. Diagram ilustrujący kontenery, w jakich rezyduje system*

# POZIOM 3: COMPONENTS – ZGRUBNA STRUKTURA

Najtrudniejszym etapem podczas tworzenia diagramu komponentów jest określenie, czym jest komponent. W przypadku każdego systemu czy każdej organizacji tworzącej oprogramowanie definicja będzie zapewne inna. Za komponent możemy uważać każdy logicznie spójne zestaw funkcjonalności, który sam w sobie posiada pewną wartość, serwis, artefakt dający się wdrożyć osobno i posiadający API.

Generalnie niemal wszyscy tworzą oprogramowanie komponentowe. Natomiast z moich obserwacji wynika, że problemy z określeniem, czym jest komponent, problemy ze zbyt ziarnistymi lub zbyt dużymi komponentami są symptomem większego problemu: braku pełnego zrozumienia struktury problemu, jakie próbujemy rozwiązać. Z tego poziomu wynikają również znacznie poważniejszej problemy na poziomie klas.

### Cel

Diagram komponentów powinien dać odpowiedź na następujące pytania:

- Z jakich komponentów/usług składa się system i jak ze sobą współpracują
- W jaki sposób system działa na poziomie komponentów
- Czy każdy komponent ma swój „dom" – czyli kontener
- W jaki sposób możemy testować automatycznie ten komponent

### Struktura

Diagram komponentów ilustruje mapowanie komponentów na kontenery. W specyficznych przypadkach jeden komponent może być wdrażany na kilka (różnych) kontenerów.

W przypadku większych i bardziej złożonych systemów możemy przyjąć regułę: na jednym diagramie komponentów przedstawiamy komponenty z jednego kontenera. Czasem bywa jednak ona ograniczająca pod względem przekazu informacji.

Dla każdego komponentu warto podać jego logiczna nazwę, technologię implementacji oraz wysokopoziomową listę odpowiedzialności.

Opisując interakcje, warto skupić się na ich celu i stylu (np. synch/asynch).

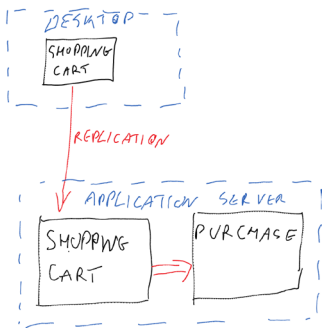
### Motywacja i Odbiorcy

Diagram tego poziomu służy zespołowi developerskiemu wytwarzającemu dany komponent w celu ilustracji: zależności i wysokopoziomych zależności. Pozwala również na wysokopoziomowe zarządzanie szacowaniem i dostarczaniem komponentów.

### Bonus: Moduły

Chwilowo zapomnijmy o tym, że moduły są tak samo rozmytym pojęciem jak komponenty oraz to, że często modułowe rozwiązania są modułowe jedynie w pojęciu handlowym (można kupić wybrane moduły, dokupić kolejne, ale „pod spodem" jest i tak wspólne spaghetti).

W systemach większej skali może być potrzebny dodatkowy poziom organizacji: moduły składające się z komponentów. Komponenty współpracują w obrębie modułu, jednak moduł oferuje i wymaga interfejsów wyższego rzędu abstrakcji niż jego poszczególne komponenty.




*Rysunek 3. Diagram ilustrujący kilka wybranych komponentów systemu i ich mapowanie na 2 kontenery (komponent Shopping Cart jest wdrażany na 2 kontenery)*

## POZIOM 4: CLASSES – TU MIESZKAJĄ SMOKI

Większość dokumentacji architektonicznej, z jaką się spotykam, oscyluje na poziomie architektury systemowej. Na diagramach tego poziomu pojawiają się czasem prostokąty z opisem „logika", „serwisy" itd., jednak nie przywiązujemy należytej wagi do porządkowania złożoności na tym poziomie.

Jednak właśnie na poziomie architektury aplikacyjnej (struktury kodu źródłowego) mieszka większość problemów:

- niezrozumienie wymagań funkcjonalnych
- nieaktualny model problemu, nad jakim pracujemy
- hackowanie nieadekwatnego modelu
- problemy z reakcją na zmiany
- problemy z rozbudową
- błędy
- problemy z testowaniem automatycznym
- część problemów wydajnościowych
- problemy z tak zwanym utrzymaniem
- to tutaj właśnie mieszkają nasze koszmary

Te małe prostokąty nazywają się krainą Legacy.

Idąc dalej: odpowiednio przygotowana struktura aplikacji (z abstrakcjami w odpowiednich miejscach) pozwala na:

- bardziej swobodne zmiany na poziomie architektury systemowej (np. zmianę granic komponentów lub wdrażanie na innego rodzaju kontenery)
- tak zwaną produktyzację oprogramowania: dostosowywanie go do potrzeb kolejnych wdrożeń

### Cel

Diagram klas ma na celu przede wszystkim zilustrowanie struktury tych klas. Powinien on komunikować styl architektoniczny obrany na poziomie arch. aplikacji, np.: warstwowy (ile i jakich warstw), heksagonalny, pipes&filers...

Odpowiedzialność każdej ze struktur (np. warstwy) powinna być ściśle określona, co przekłada się na jasne wytyczne podczas projektowania konkretnej funkcjonalności i tworzenia kodu źródłowego.

### Struktura

Struktura graficzna tych diagramów powinna odzwierciedlać dokładnie strukturę logiczną (np. warstwy) oraz mieć dosłowne przełożenie na fizyczne struktury kodu (np. pakiety).

Często lepszym podejściem niż rysowanie konkretnych klas z systemu jest przedstawienie wzorcowego projektu. Na takim diagramie znajdują się wówczas przykładowe klasy jako „proof of concept", a zadaniem programistów jest wzorowanie się na tej strukturze podczas tworzenia własnych klas.

Dodatkowo dla każdej z warstw (lub innych struktur) dobrze jest określić archetypowe Building Blocks, jakie mogą być używane na danym poziomie. Warto skorzystać tutaj ze sprowadzonych Building Blocks takich jak: Responsibility Driven Design oraz Wzorce Taktyczne z puli Domain Driven Design.

Oczywiście różne komponenty mogą być wykonane wg różnych architektur aplikacyjnych – w zależności od potrzeb.

### Motywacja

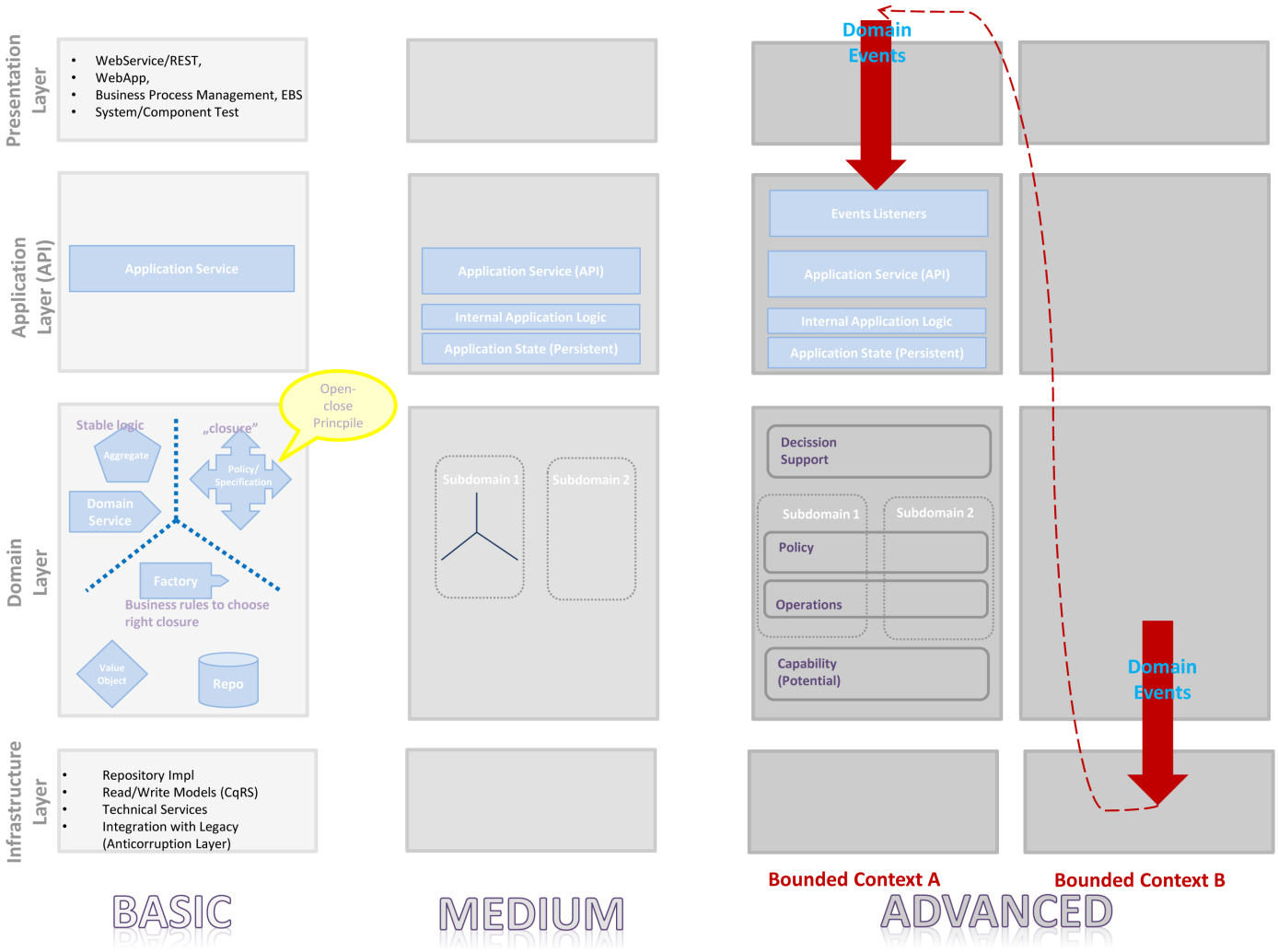
Struktury aplikacyjne pozwalają wprowadzić porządek na poziomie kodu źródłowego. Diagramy tego typu są doskonałą „mapą" dla programistów oraz przewodnikiem dla nowych członków zespołu. W większych projektach warto przygotować kilka archetypowych architektur aplikacyjnych i dobierać je w zależności od stopnia złożoności.

### Odbiorcy

Odbiorcą są programiści i projektanci pracujący nad konkretnymi komponentami.

Rysunek 4 przedstawia przykład 3 podejść do architektury warstwowej. Diagram tego typu narzuca pewną strukturę. Uzupełnieniem diagramu powinien być opis archetypowej struktury zawierający specyfikację odpowiedzialności każdej z warstw.

W pierwszej z nich rozdzielmy logikę aplikacyjną od domenowej oraz separujemy kod stabilny od tego, który jest często poddawany zmianom (ew. służy do kastomizacji). Kolejna wersja tej architektury wyróżnia poddomeny w modelu. Najbardziej zaawansowana wersja wyróżnia 4 poziomy modelu




*Rysunek 4. Diagram ilustrujący przykładowe struktury dla klas – 3 poziomy rozbudowy*

oraz poddomeny. Szczegółowy opis tych architektur wraz z odpowedzialnościami każdej ze struktur można znaleźć w serii artykułów poświęconych Domain Driven Design (ramka).

## PROCES: PODEJŚCIE ODDOLNE

Podejście 4C pomaga nam strukturyzować wiedzę, nadając ramy na poziomy ogólności. Jest to również naturalny proces wdrażania się w istniejący system.

Oczywiście rozpoczynając pracę nad projektem, nie jesteśmy zwykle w stanie z góry określić, jakich kontenerów potrzebujemy oraz jakie komponenty będą istnieć w systemie. Podobnie struktury klas klarują się wraz z wprowadzaniem złożoności i rozbudową (jednak warto znać te struktury, aby wiedzieć, w jaki sposób w ogóle można wprowadzać uporządkowanie).

Dlatego w zwinnym projekcie sprawdza się raczej podejście oddolne.

1. Rozpoczynamy z poziomu Context, aby określić ramy
2. W pierwszych iteracjach skupiamy się na wyłanianiu struktur na poziomie Classes
3. Następnie wyłaniają się Componens
4. Nadchodzi odpowiedni moment na zastanowienie się nad rozmieszczeniem Components w Containers

### Diagramy w podejściu 4C

- Context – ogólny kontekst, w jakim będzie pracował system: główni aktorzy/role, inne systemy, główne procesy
- Containers – wysokopoziomowe decyzje, „domki" dla komponentów, wysokopoziomowa komunikacja, informacja dla developerów o tym, gdzie należy wykonać pracę
- Components – logicznie spójne części funkcjonalności (np. serwisy), działanie systemu na poziomie przepływu pomiędzy komponentami
- Classes – struktury dla klas, architektura aplikacji (porządkowanie kodu źródłowego), zaniedbania na tym poziomie to katastrofa całego przesięwzięcia

## WERYFIKACJA: ZOOM-IN I ZOOM-OUT

Podejście 4C zapewnia nam spójne „skalowanie informacji" bez luk semantycznych. Dzięki temu możemy weryfikować architekturę – w sensie aktualnej jej zbieżności z wymaganiami i kontekstem. Weryfikację ułatwia możliwość płynnego skalowania uwagi od ogółu do szczegółu, co pozwala stopniowo „ładować kontekst".

**Sławomir Sobótka** slawomir.sobotka@bottega.com.pl

Programujący architekt aplikacji specjalizujący się w technologiach Java i efektywnym wykorzystaniu zdobyczy inżynierii oprogramowania. Trener i doradca w firmie Bottega IT Solutions. W wolnych chwilach działa w community jako: prezes Stowarzyszenia Software Engineering Professionals Polska (http://ssepp.pl), publicysta w prasie branżowej i blogger (http://art-of-software.blogspot.com).