Wykład 3

Zasady projektowania pakietów i komponentów

PP – Package Principles

Zasady projektowania pakietów i komponentów (*PP – Package Principles*)

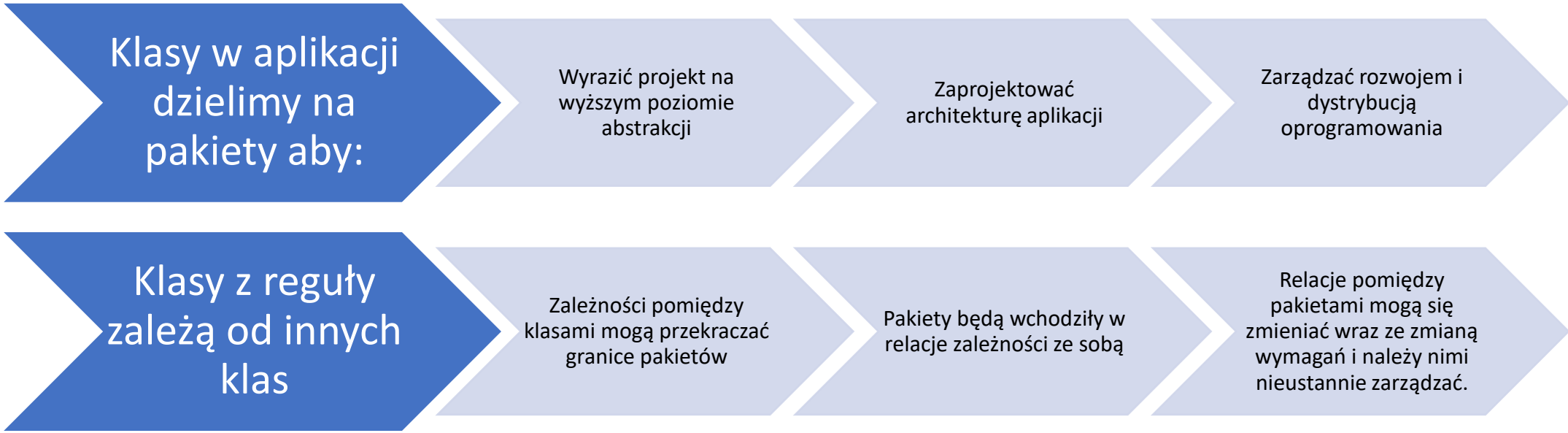
W paradygmacie obiektowym

- Stosujemy zasady **SOLID**

W paradygmacie komponentowym

- Stosujemy **Zasady projektowania pakietów i komponentów (PP - Package Principles)**
  - Ziarnistość (spójność pakietów)
  - Stabilność (sprzężenia - zależności pakietów)

Zasady projektowania pakietów i komponentów (*PP – Package Principles*)

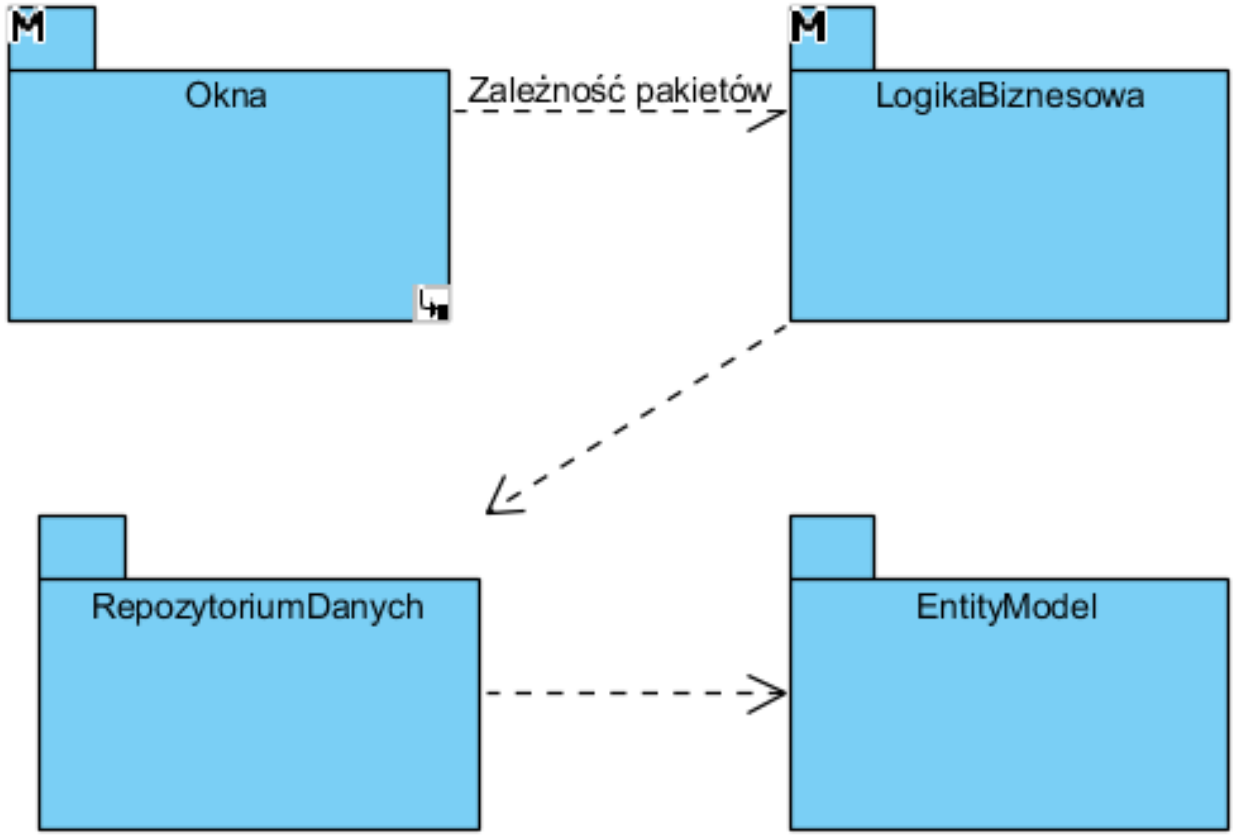


Podział aplikacji na wyższym poziomie abstrakcji

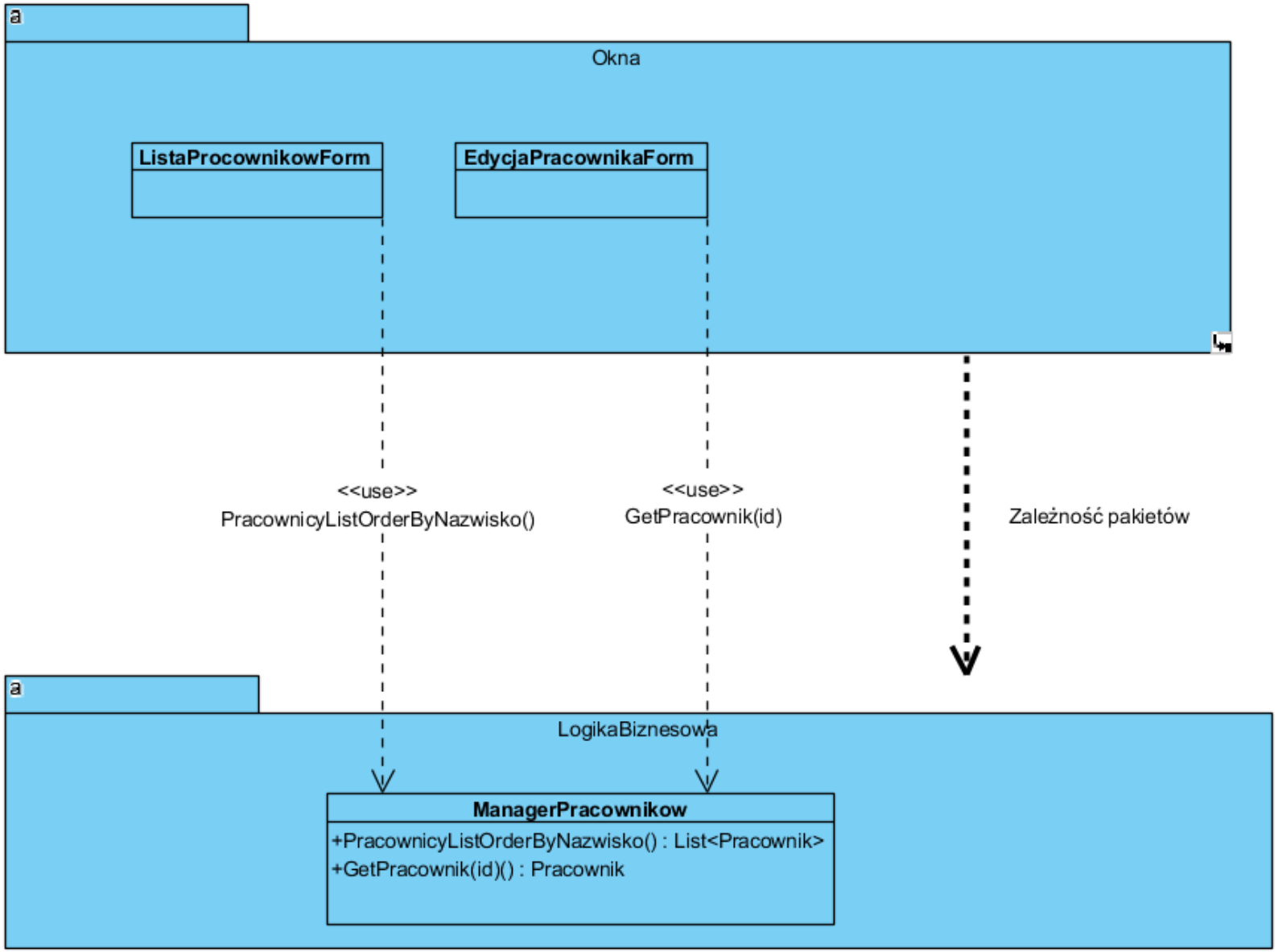
Przykład aplikacji podzielonej na warstwy

- Okna (GUI)
- LogikaBiznesowa
- RepozytoriumDanych
- EntityModel

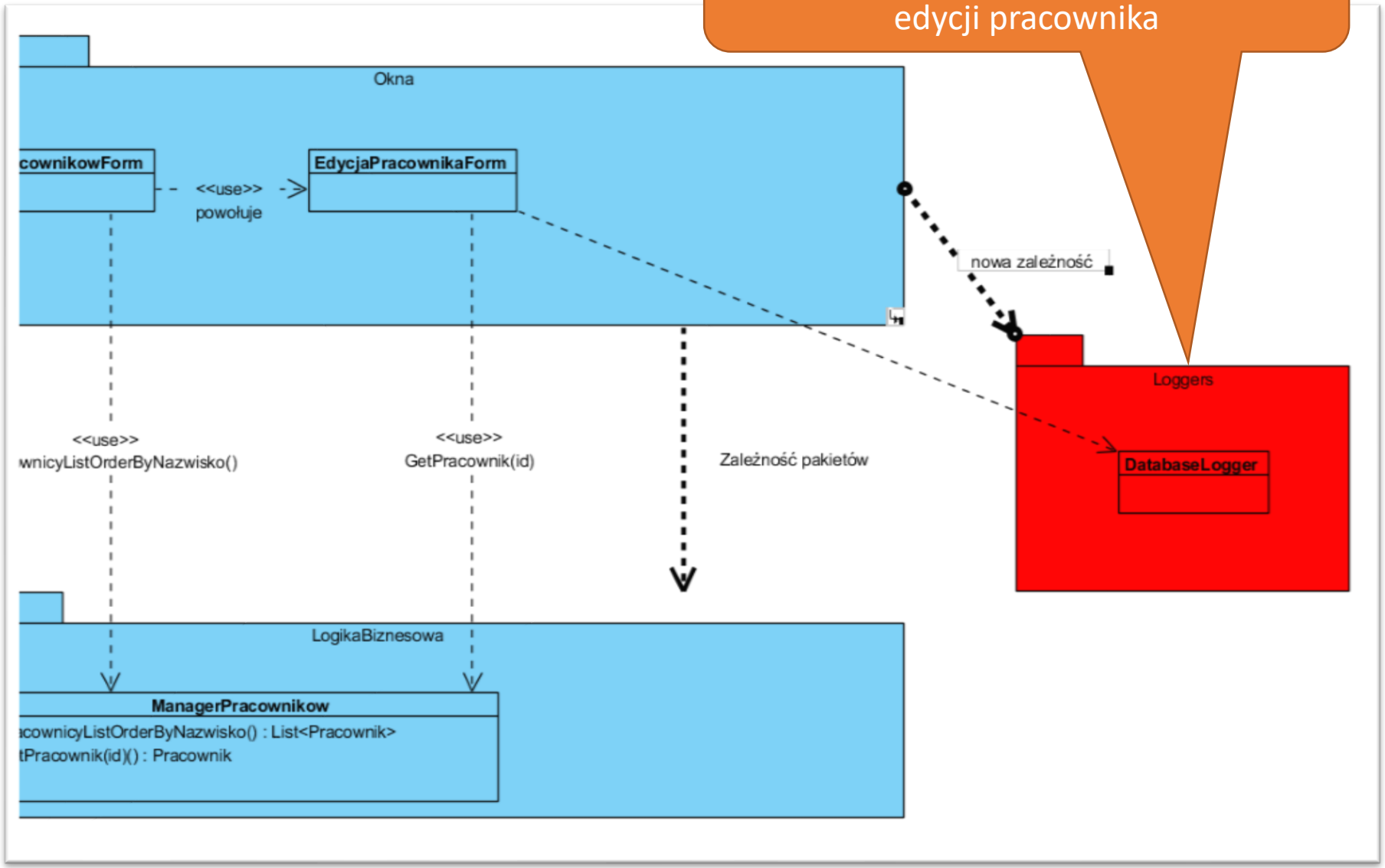
Diagram pakietów UML



Zależności pomiędzy klasami mogą przekraczać granice pakietów



Relacje pomiędzy pakietami mogą się zmieniać wraz ze zmianą wymagań i należy nimi nieustannie zarządzać



Zasady projektowania pakietów i komponentów (*PP – Package Principles*)

Zasady PP – dzielimy na dwie kategorie

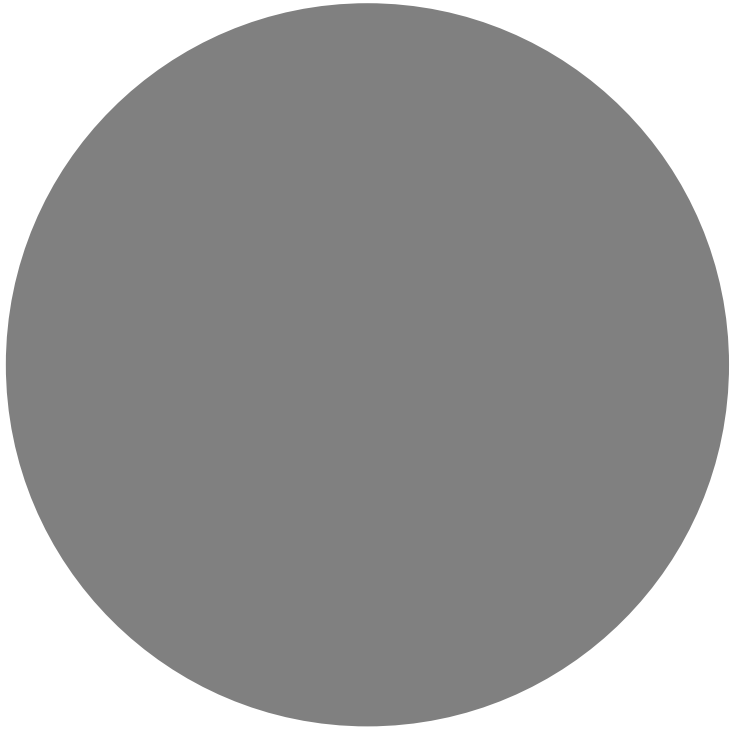
Zasady spójności pakietów

- określają jak przydzielać klasy do pakietów
  - REP (Reuse/Release Equivalence Principle)
  - CRP (*Common Reuse Principle*)
  - CCP (*Common Clousure Principle*)

Zasady stabilności pakietów

- określają wzajemne zależności pomiędzy pakietami oraz definiują pewne metryki opisujące stabilność pakietów.
  - ADP(Acyclic Dependecies Principle)
  - SDP (Stable Dependecies Principle)
  - SAP (Stable Abstractions Principle)

Zasady spójności komponentów



Ziarnistość komponentów

Zasady spójności komponentów

Ponowne użycie kodu

Kodu nie używamy poprzez kopiowanie z innej klasy do naszej

Jeśli autor oryginału poprawia błędy czy dodaje nowe funkcjonalności, nasza klasa nie widzi tych zmian

Programista sam musi znaleźć to co się zmieniło i zaktualizować swój kod

Twój kod i kod oryginału nigdy nie będzie spójny

Zasady spójności komponentów

Ponowne użycie kodu

Zamiast kopiowania, kod powinien być ponownie użyty poprzez dołączenie do programu skompilowanej biblioteki

Programista nie musi widzieć ani mieć dostępu kodu źródłowego, żeby go użyć.

Autor biblioteki, jest odpowiedzialny za wsparcie i rozwijanie kodu biblioteki

Zasady spójności komponentów - REP *(ang. REP* - Reuse-Release Equivalence Principle*)*

Zasada równoważności wielokrotnego użycia i wydawania(dystrybucji)

Komponent musi zawierać klasy ponownego użytku

- Zawiera klasy z tej samej rodziny
- Klasy niezwiązane z przeznaczeniem komponentu nie mogą być w nim zawarte.

Komponent złożony z rodziny klas wielokrotnego użycia staje się bardziej użyteczny i re-używalny

Zasady spójności komponentów - REP *(ang. REP* - Reuse-Release Equivalence Principle*)*

Zasada równoważności wielokrotnego użycia i wydawania(dystrybucji)

Wydawca komponentu musi:

- Informować o planowanych zmianach w interfejsach i funkcjonalnościach kodu
- Zapewnić wersjonowanie komponentów, pakietów
- Umożliwić wsparcie dla nowych oraz starszych wersji

Jednostka wielokrotnego użycia jest jednostką dystrybucji

- Każda zmiana funkcjonalności czy poprawa błędów powinna iść w parze ze zmianą wersji dystrybucyjnej
- Dobór klas w komponencie powinien się odbywać z uwzględnieniem użytkownika naszego kodu. Albo wszystkie klasy są wielokrotnego użytku albo żadna
- Kod tworzony jest dla ludzi i im ma służyć

Zasady spójności komponentów -CRP *(ang. CRP – Common Reuse Principle)*

Zasada zbiorowego wielokrotnego stosowania

Ułatwia zarządzanie kolekcją klas, które powinny znaleźć się w jednym komponencie

- Klasy, które będą wielokrotnie stosowane w sposób łączny powinny należeć do tego samego komponentu np.:
  - System.Windows.Forms.dll
  - Mscorlib.dll
- Klasy tego samego pakietu są ze sobą w ścisłych relacjach, łączy je wiele zależności

CRP mówi również, których klas nie należy umieszczać w ramach jednego komponentu.

- Klasy które nie są ze sobą ściśle związane nie powinny należeć do tego samego komponentu

Zasady spójności komponentów -CRP *(ang. CRP – Common Reuse Principle)*

Zasada zbiorowego wielokrotnego stosowania

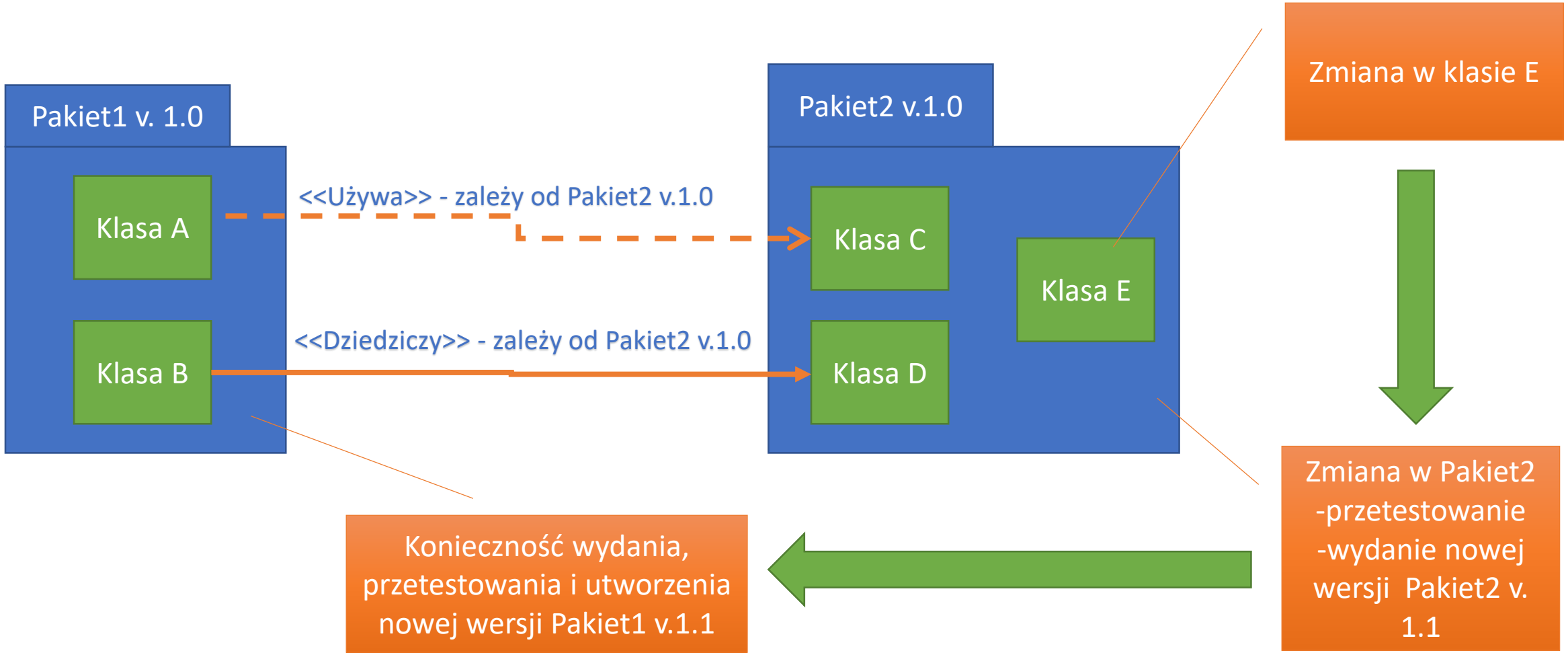
CRP mówi również, których klas nie należy umieszczać w ramach jednego komponentu.

- Gdy pakiet A zależy tylko od jednej klasy pakietu B to pomiędzy pakietami jest zależność. A->B
- Jeżeli w pakiecie B zmienimy inną klasę, od której pakiet A nie zależy to:
  - A dalej zależy od B
  - Pakiet B musi być ponownie wydany – nowa wersja
  - Pakiet A również musi być ponownie przetestowany i wydany ze względu na zmianę jego zależności.

Taka sytuacja wymusza niepotrzebną pracę

- Ponowną dystrybucję pakietu A
- Konieczność przetestowania pakietu A

Zasady spójności komponentów -CRP *(ang. CRP – Common Reuse Principle)*



Zasady spójności komponentów - CPP *(ang. Common Clousure Principle)*

Zasada zbiorowego zamykania

Odpowiednik zasady SRP (pojedynczej odpowiedzialności) tylko że dla komponentu.

Klasy wchodzące w skład komponentu powinny być zbiorczo zamknięte na określony rodzaj zmian.

Zmiana, która ma wpływ na komponent, ma wpływ także na wszystkie jego klasy, ale nie na pozostałe komponenty

Zasada nakazuje ograniczenie potencjalnych źródeł zmian na poziomie komponentu

Zasady spójności komponentów - CPP *(ang. Common Clousure Principle)*

Zasada zbiorowego zamykania

- W większości przypadków zależy nam bardziej na możliwości utrzymania aplikacji niż na możliwości wielokrotnego użycia
- W przypadku zmiany wymagań lub zmian w klasach nie chcemy aby te zmiany dotyczyły wielu pakietów tylko najlepiej jednego
- Jeżeli zmiany dotyczą jednego pakietu, to tylko ten pakiet musimy ponownie dystrybuować

Nie da się zamknąć pakietu na wszelkie zmiany

- Pakiet domykamy na zmiany których doświadczyliśmy
- W jednym pakiecie powinny być klasy, które są otwarte na określony rodzaj zmian
- Jeżeli taka zmiana wystąpi, to jest szansa, że obejmie niewielką liczbę pakietów.

Zasady stabilności komponentów

Stabilność konponentów

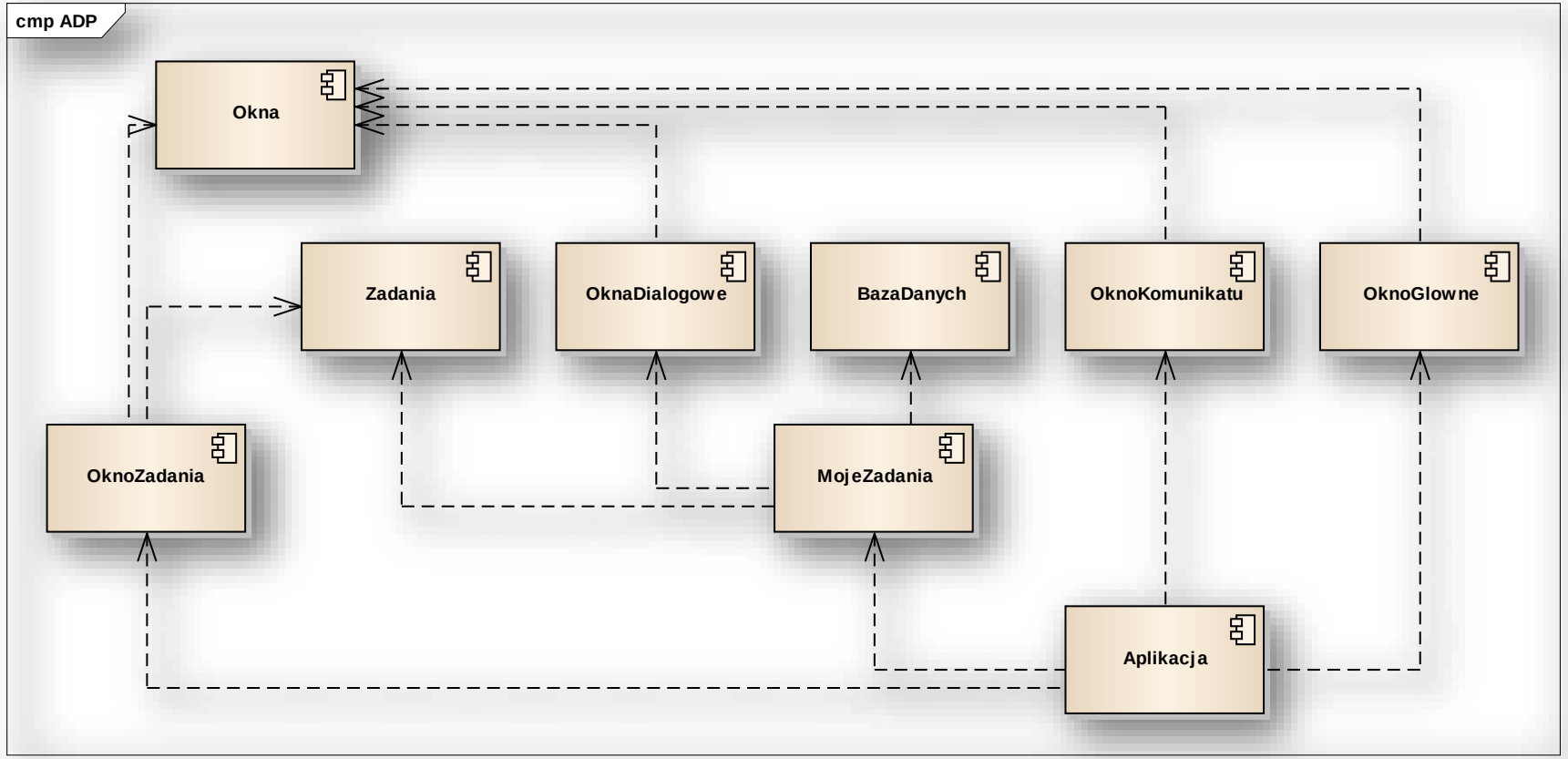
Zasady stabilności komponentów – ADP *(ang.* Acyclic Dependecies Principle*)*

Zasada acyklicznych zależności

W grafie zależności pomiędzy komponentami nie mogą występować żadne cykle

Zasady stabilności komponentów – ADP *(ang.* Acyclic Dependecies Principle*)*

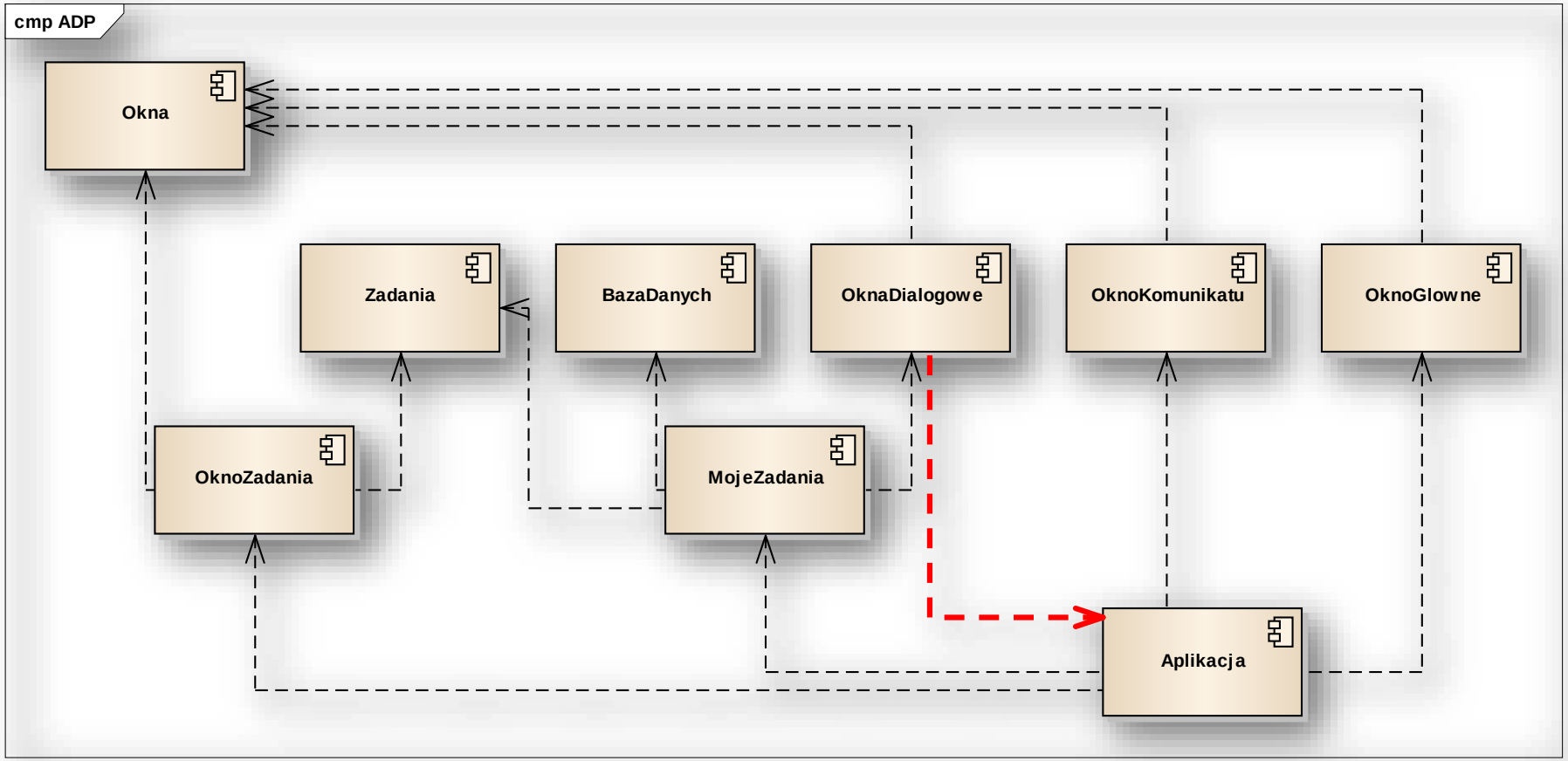
Wyobraźmy sobie projekt z następującymi komponentami



Zasady stabilności komponentów – ADP *(ang. Acyclic Dependecies Principle)*

Załóżmy pewną zmianę w projekcie:

- Jedna z klas komponentu **OknaDialogowe** wykorzystuje jedną z klas komponentu **Aplikacja**



Zasady stabilności komponentów – ADP *(ang.* Acyclic Dependecies Principle*)*

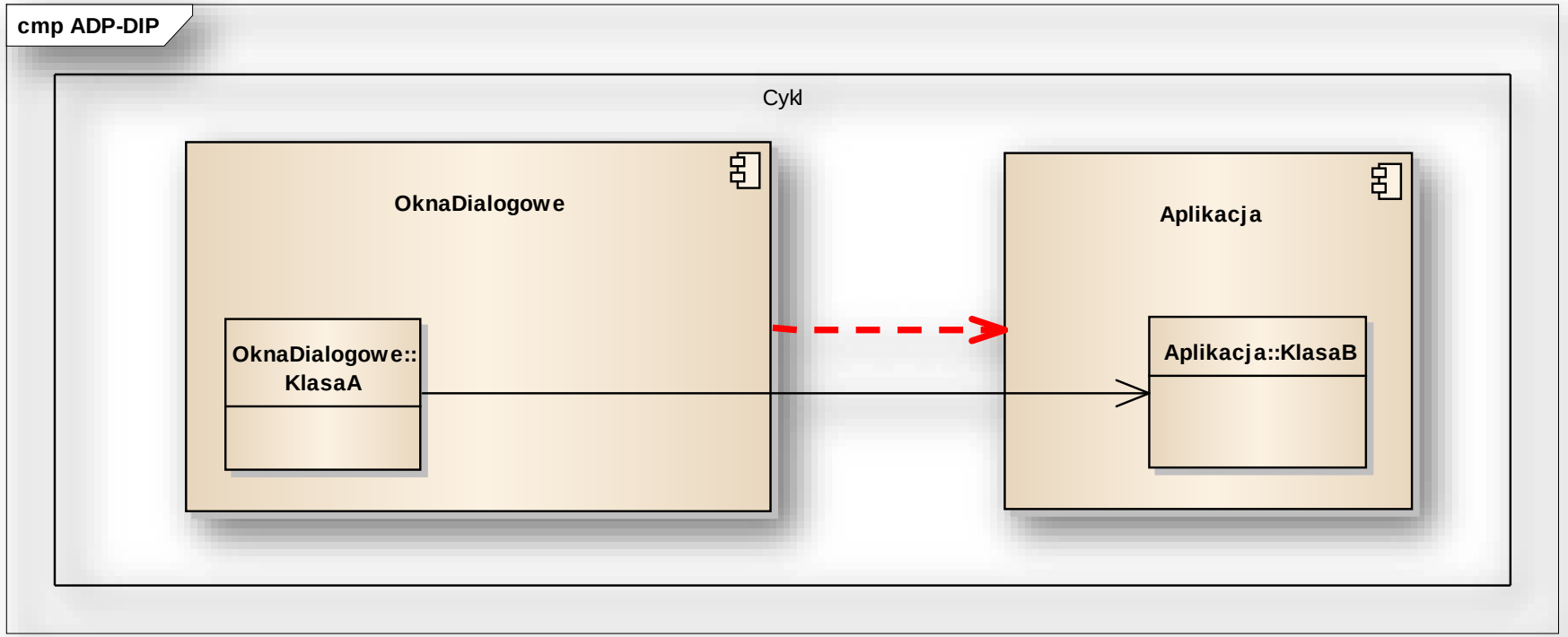
Wprowadznie cyklu w grafie zalezności niesie za sobą poważne problemy:

- Komponent **MojeZadania** zależy od wszystkich komponentów systemu !!!
- Komponenty MojeZadania, OknaDialogowe, Aplikacja muszą być wydawane razem ze względu na ich wzajemną zależność
- Zmiany w którymkolwiek z tych komponentów mogą wpłynąć na działanie pozostałych
- Praktycznie mamy do czynienia z jednym wielkim komponentem.

Zasada acyklicznych zależności (ADP – Acyclic Dependecies Principle)

- Zastosowanie zasady DIP (Dependency Inversion Principle)
- Utworzenie nowego komponentu z klasą(ami) od których zależą obydwa komponenty

Jak przerwać cykl używając DIP?



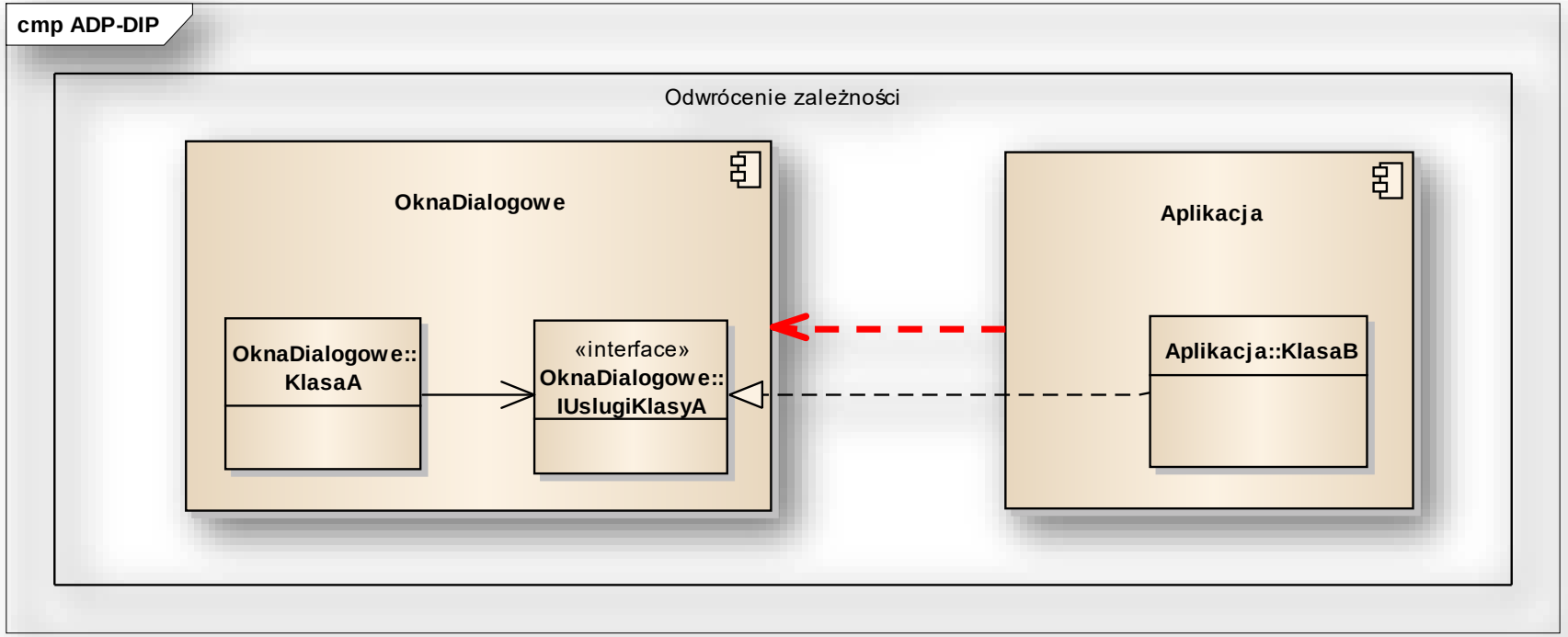
- Musimy się pozbyć zależności klasyA od KlasyB
- W zasadzie musimy tę zależność odwrócić

Jak przerwać cykl używając DIP

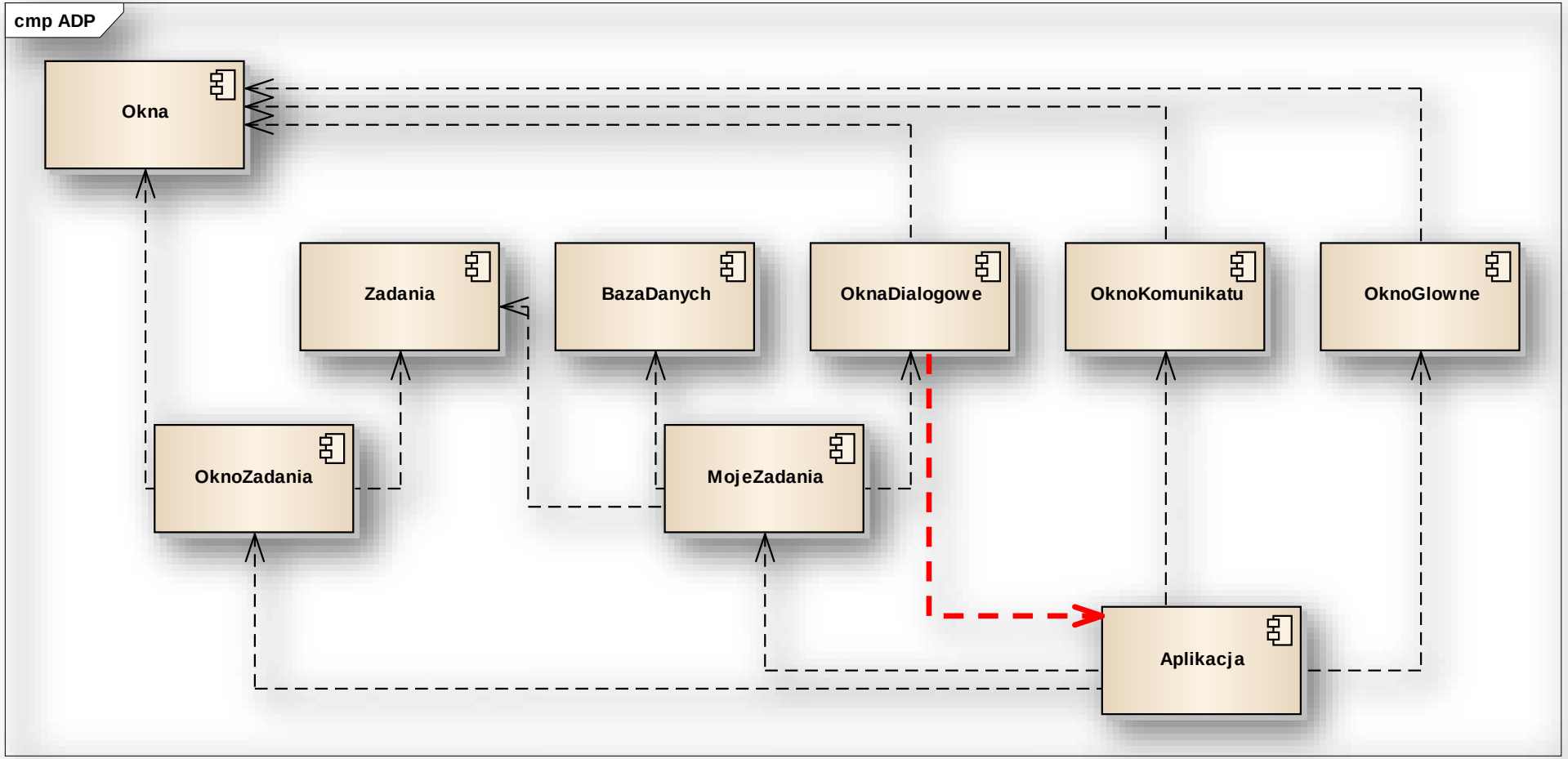
Odwrócić zależność wprowadzając interfejs usług klasy KlasaA
w komponencie OknaDialogowe

KlasaA staje się zależna od interfejsu **IUslugiKlasyA** implementowanego przez **klasaB** a nie od samej klasy **klasaB**

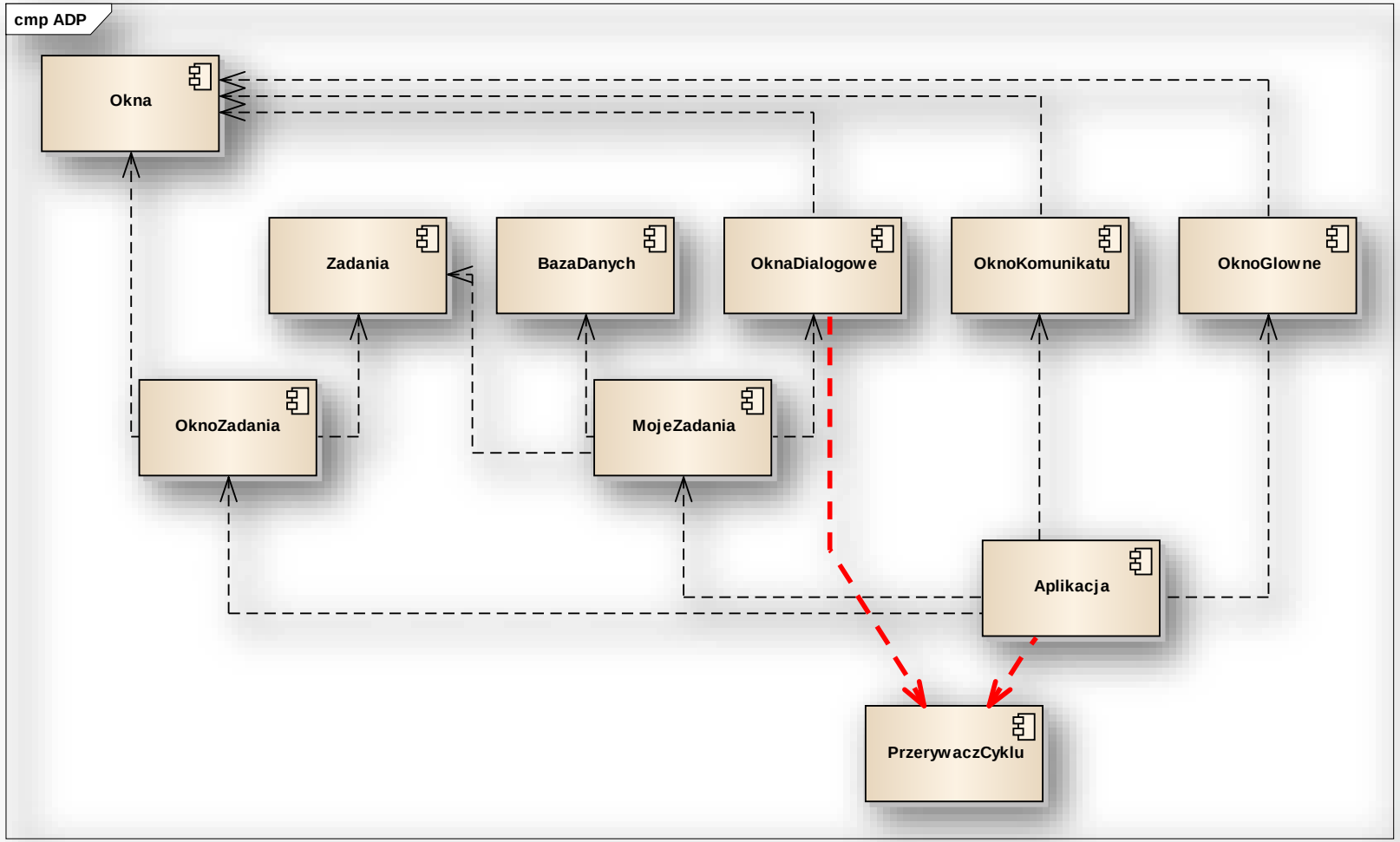
Tym samym komponent **Aplikacja** staje się zależny od komponentu **OknaDialogowe**



Jak przerwać cykl używając nowego komponentu?



Jak przerwać cykl używając nowego komponentu?



Zasady stabilności komponentów – SDP *(ang. Stable Dependecies Principle)*

Zasada stabilnych zależności

Stabilność

- czy człowiek stojący na jednej nodze jest stabilny?
  - Niby nie….
  - Ale gdyby go nie ruszać? To może tak….

Stabilny komponent

- Taki, którego trudno modyfikować ze względu na dużą ilość zależności przychodzących

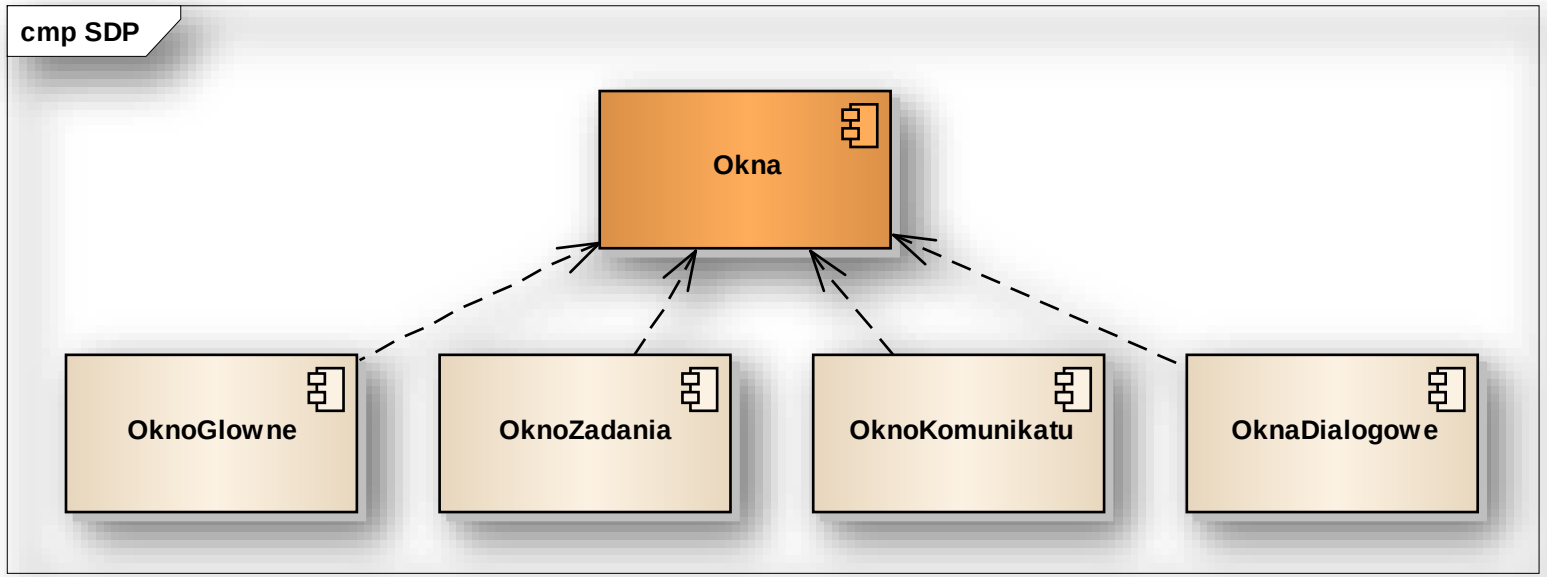
Niestabilny komponent

- Taki, którego modyfikacja nie wpływa na inne komponenty ze względu na brak zależności przychodzących

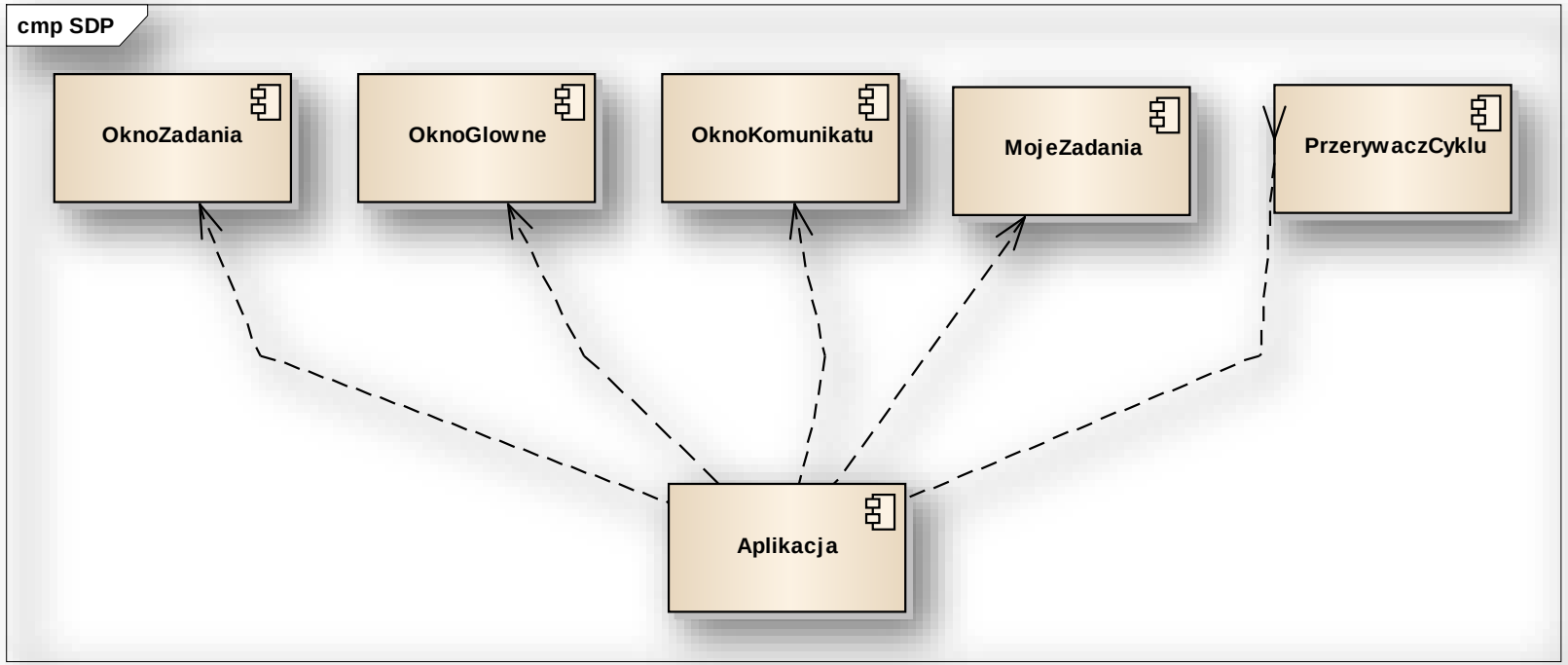
Zasady stabilności komponentów – SDP *(ang. Stable Dependecies Principle)*

Komponent stabilny „Okna”

- Zależą od niego pozostałe komponenty
- Jego modyfikacja wymagałaby sprawdzenia, przetestowania i wydania zależnych komponentów



Zasady stabilności komponentów – SDP *(ang. Stable Dependecies Principle)*



Komponent niestabilny „Aplikacja”

- Zależy od pozostałych komponentów
- Jego modyfikacja nie wpływa na pozostałe komponenty
- Jest podatny na zmiany

Miara stabilności

Jak mierzyć stabilność komponentów?

Stabilność pozycyjna

stosunek zależności wychodzących do sumy zależności przychodzących oraz wychodzących

$$I = \frac{Ce}{Ce + Ca}$$

Ca – zależności przychodzące – liczba klas spoza danego komponentu, które zależą od klas należących do tego komponentu

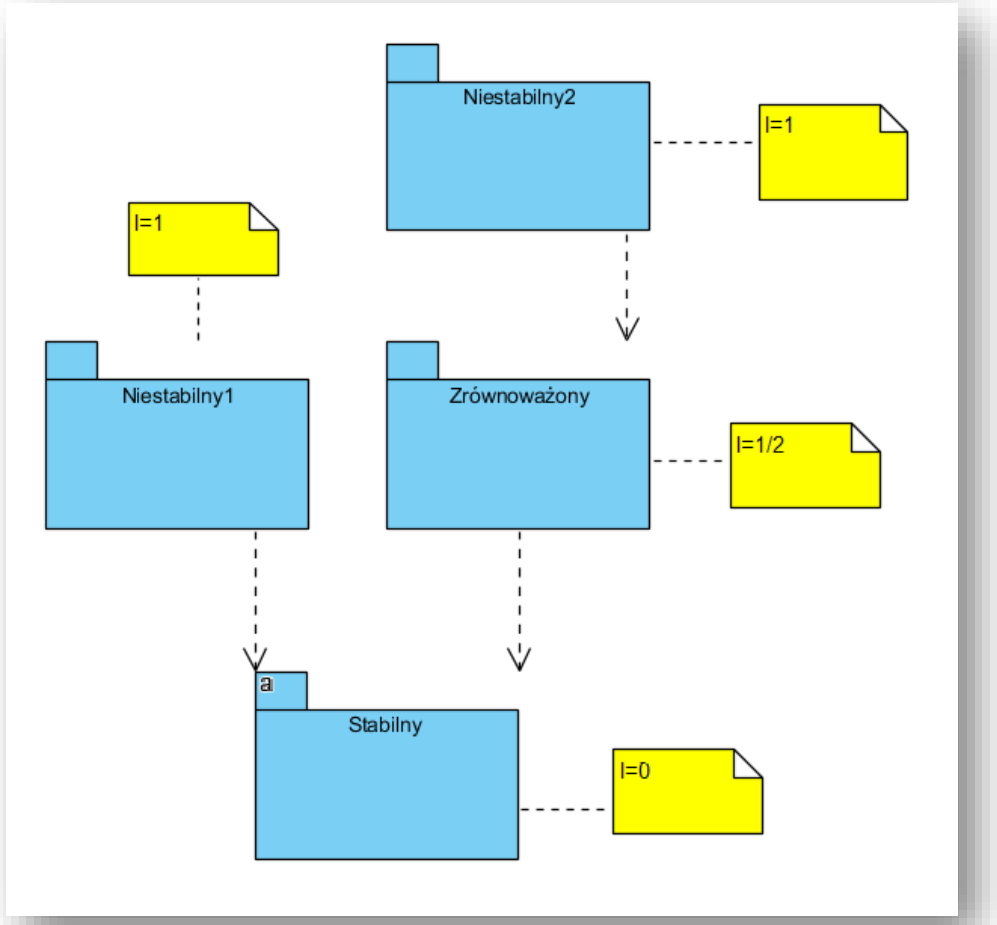
Ce- zależności wychodzące – liczba klas w ramach danego komponentu, które zależą od klas spoza niego

I – niestabilność (*ang. instability*)

Zasady stabilności komponentów – SDP *(ang. Stable Dependecies Principle)*

Zasada stabilnych zależności
(SDP – Stable Dependecies Principle)

- Mówi o tym, że metryka „I” (niestabilność) pakietu A powinna być większa niż metryka „I” pakietu B od którego A zależy
- Metryki „I” powinny maleć w kierunku zależności (wartości metryk podano w żółtych polach)



Zasady stabilności komponentów – SAP *(ang. Stable Abstractions Principle)*

Zasada stabilnych abstrakcji

Połączenie SDP i SAP to odpowiednik OCP (Zasada Otwarte/Zamknięte z SOLID) tylko w kontekście całych pakietów

W przypadku klas, było czarno-biało:

- albo klasa była abstrakcyjna albo nie.
- Komponenty mogą przyjmować wartości pośrednie.

Zasady stabilności komponentów – SAP *(ang. Stable Abstractions Principle)*

Zasada stabilnych abstrakcji

Zasada wprowadza zależność pomiędzy abstrakcyjnością i stabilnością

Komponent powinien być równie abstrakcyjny co stabilny

Komponent stabilny powinien być na tyle abstrakcyjny aby jego stabilność nie uniemożliwiała jego rozbudowy

Zasady stabilności komponentów – SAP *(ang. Stable Abstractions Principle)*

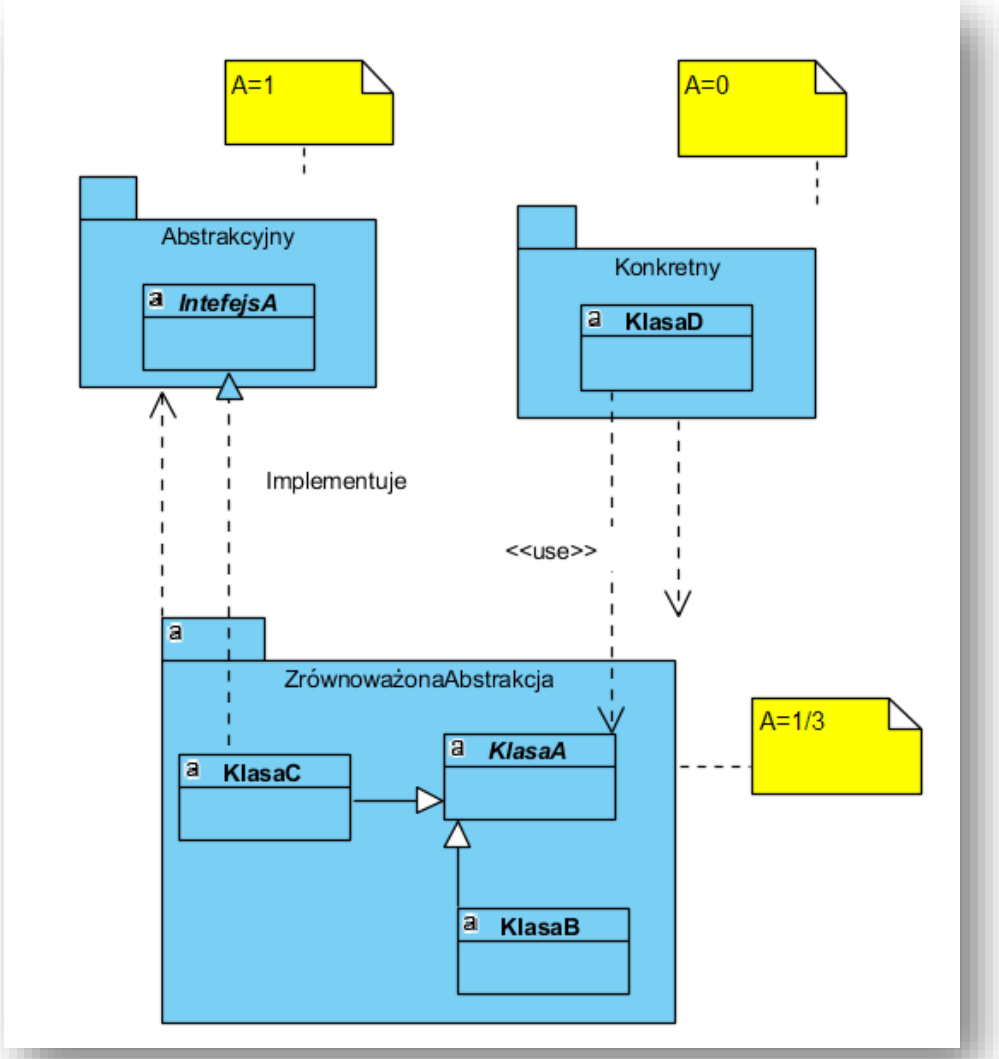
Zasada stabilnych abstrakcji

Pakiety czym bardziej abstrakcyjne powinny być jednocześnie bardziej stabilne

- A= 0,9 i I=0,9 – źle
- A= 0,9 i I=0,1 – ok

Pakiety bardziej konkretne(nieabstrakcyjne) powinny być bardziej niestabilne

- A= 0,1 i I=0,1 – źle
- A= 0,1 i I=0,9 – ok



Miara abstrakcji

Jak mierzyć abstrakcję komponentów?

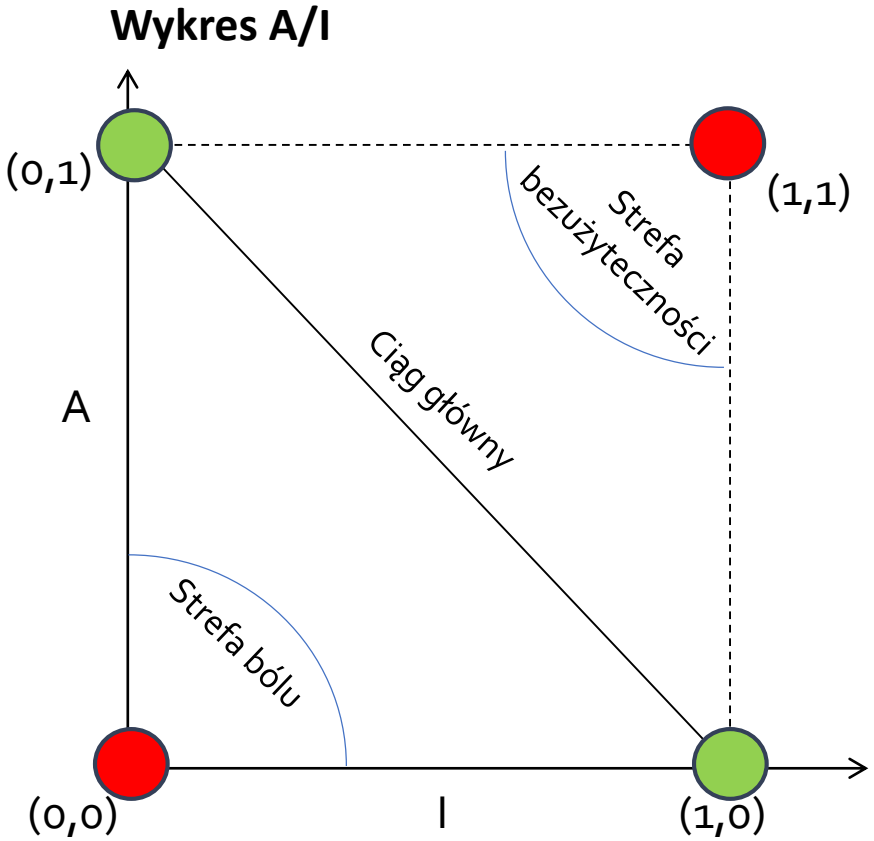
$$A = \frac{Na}{Nc}$$

Na –liczba klas abstrakcyjnych z danego komponentu

Nc –liczba wszystkich klas z danego komponentu

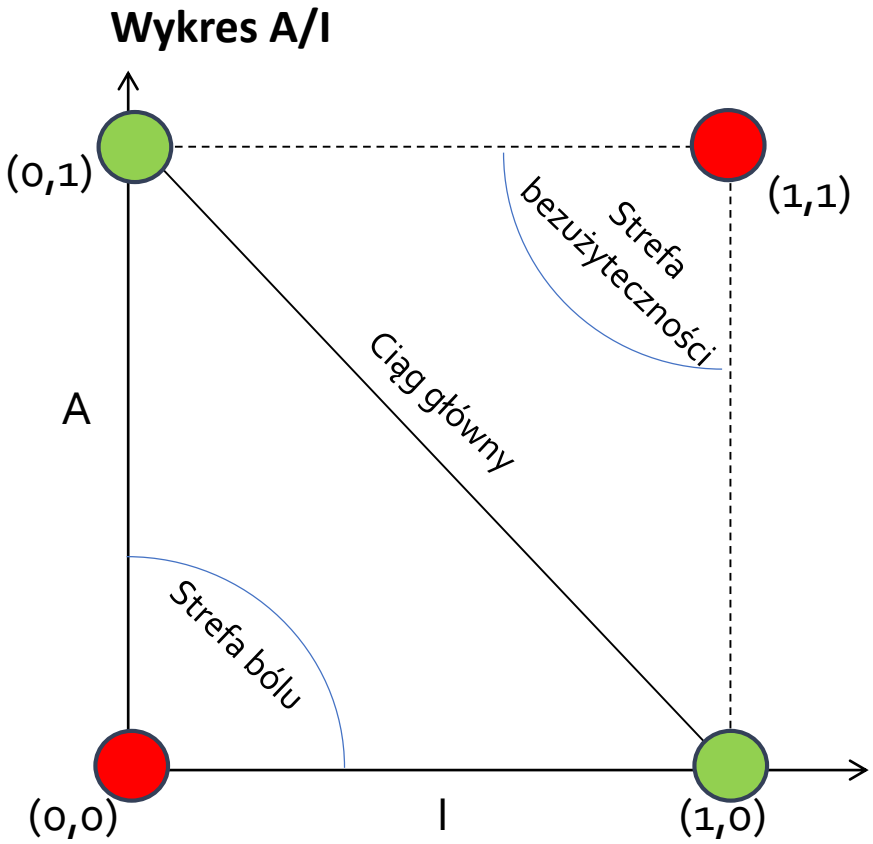
Ciąg główny

Zależność pomiędzy abstrakcyjnością a stabilnością pakietu



- A – abstrakcja (abstaction)
- I – niestabilność (Instability)
- Zielone kółka- komponenty idealne (stabilne i abstrakcyjne oraz niestabilne i konkretne)
- Czerwone kółka – komponenty nieodpowiednie
- Strefa bólu – komponenty stabilne i konkretne
- Strefa bezużyteczności – komponenty maksymalnie abstrakcyjne ale nie posiadające komponentów zależnych
- Ciąg główny – komponenty o zrównoważonej abstrakcji i stabilności

Ciąg główny



Dobra konfiguracja pakietów to taka, gdzie pakiety leżą w pobliżu ciągu głównego

- Nie każda klasa abstrakcyjna będzie maksymalnie abstrakcyjna,
- może zależeć od innej abstrakcji i wtedy jej stabilność wzrasta

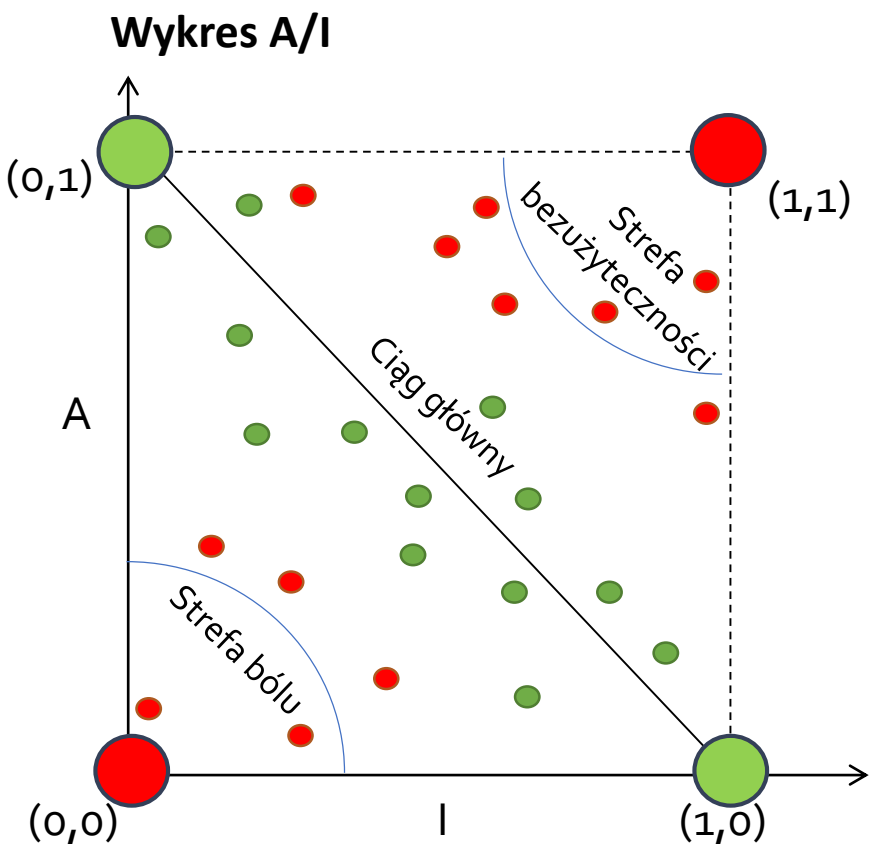
Pakiety w strefie bólu są

- sztywne konkretne i nie da się ich rozszerzać bo nie są abstrakcyjne
- Wprowadzanie zmian jest bolesne
- Czego nie dotkniesz, masz gwarancję że coś innego się posypie ☹

Pakiety w strefie bezużyteczności są

- maksymalnie abstrakcyjne i nic od nich nie zależy
- więc po co nam takie?
- robienie abstrakcji na zapas to marnotrawstwo cennej pracy programisty

Ciąg główny



Dobry projekt

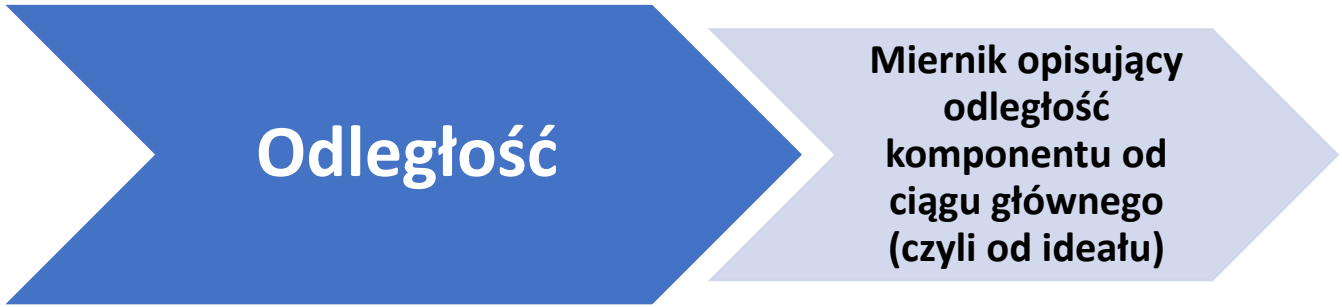
- Małe odległości pakietów od ciągu głównego

Zły projekt

- Duże odległości pakietów od ciągu głównego

Odległość od ciągu głównego

Jak mierzyć stabilność komponentów?



$$D = \frac{|A+I-1|}{\sqrt{2}}$$

$$D' = |A + I - 1|$$

D - odległość przyjmuje wartości z zakresu [0, 0.707]

D' – odległość znormalizowana przyjmuje wartości z zakresu [0,1]

EOL=true;

Proszę się rozejść ☺

Do pooglądania i poczytania

- Robert C. Martin
- https://vimeo.com/68236438

- **Robert C. Martin**
- **„Zwinne wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki”, Helion 2015,2017**