Architektura aplikacji w języku C

Architektura fizyczna i logiczna

- Język C oraz C++, to jedne z tych obecnych na rynku które wymagają rozróżnienia pomiędzy architekturą logiczną projektu, a jego architekturą fizyczną:

    - **Architektura logiczna** - obejmuje elementy logiczne oraz relacje je łączące. Bezpośrednio realizuje dostarczanie wartości dla klienta i zaspokajanie jego wymagań. Nazewnictwo i rola elementów tej architektury, bezpośrednio odnosi się do pojęć domenowych.

    - **Architektura fizyczna** - obejmuje elementy fizyczne projektu oraz relacje między nimi. Pliki źródeł, pliki nagłówków, skompilowane pliki obiektów *.o/*.obj/*.a/*.dll/*so/… Nie wszystkie odnoszą się bezpośrednio do problemów domeny.

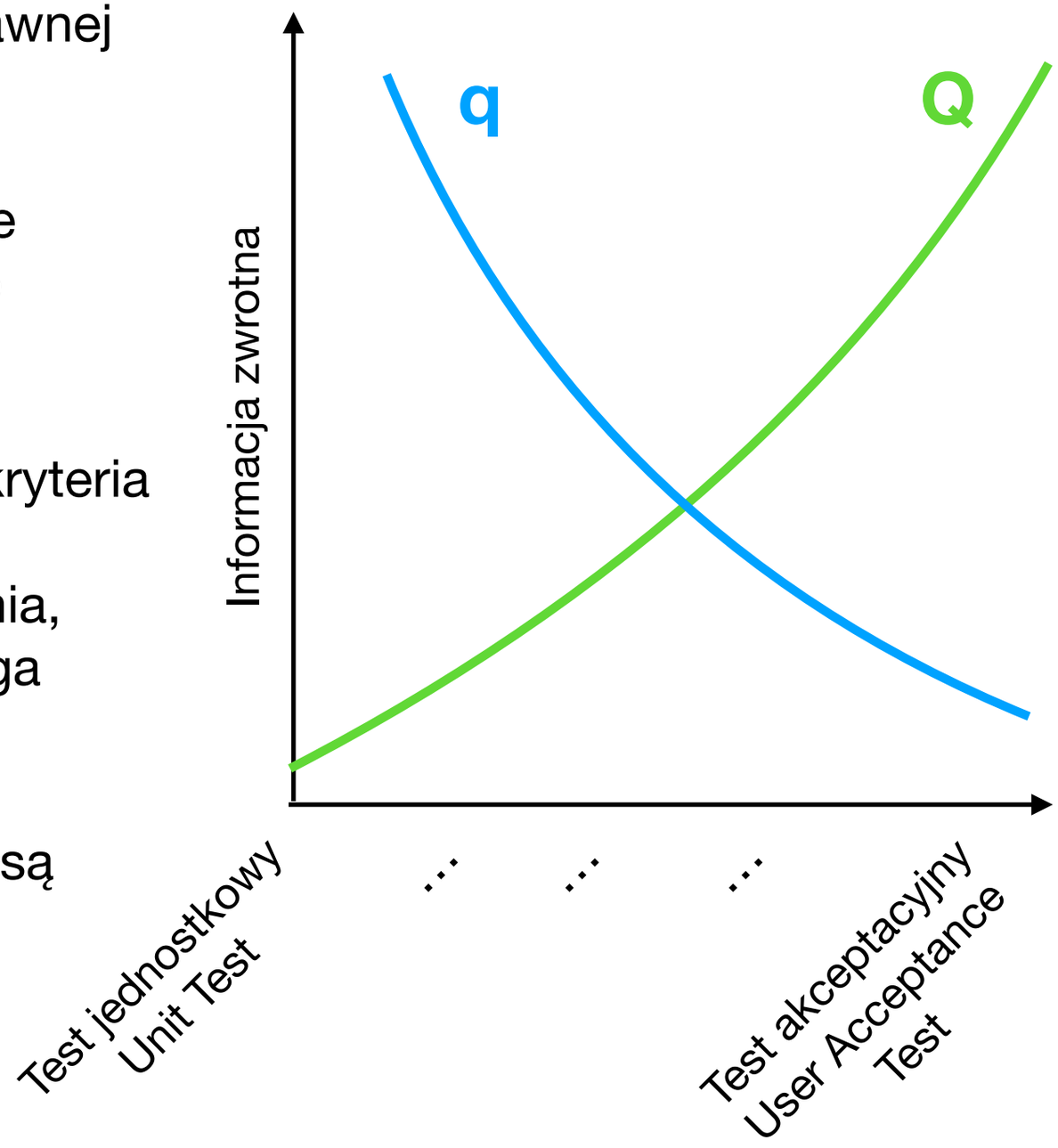
- Architektura logiczna wpływa na możliwości rozszerzenia funkcjonalności aplikacji.

- Architektura logiczna nie może być jednak rozwijana przy niskiej jakości architektury fizycznej projektu.

Dwa oblicza jakości — model uproszczony

- Projekt realizowany jest przy ocenie (jawnej bądź nie), dwóch rodzajów jakości:
  - **Q** - jakość zewnętrzna. Obejmuje te kryteria które klient poddaje ocenie rozważając wartość aplikacji.
  - **q** - jakość wewnętrzna. Obejmuje kryteria związane ze strukturą aplikacji i jej komponentów, łatwości jej rozwijania, możliwości rozszerzenia itp. Podlega ocenie przez personel techniczny.
- W praktyce obydwie jakości oceniane są głównie z użyciem testów.

Czy te jakości wchodzą ze sobą w korelację? Jeśli tak to jaką?



Brak korelacji jakości?

- Jest to chyba znana (i co zaskakujące) racjonalna praktyka obniżania jakości wewnętrznej (q) kosztem utrzymania … terminu dostarczenia produktu a więc … jakości zewnętrznej (Q).

- Jednak w przypadku drastycznego obniżenia jakości q, rozwój produktu będzie spowolniony lub sparaliżowany.

- Dzieje się tak ze względu na powstawanie **długu technologicznego**.

- Trzeba podkreślić że **w każdym projekcie dług technologiczny powstaje** (aplikacje R&D, startup - spory … systemy krytyczne - niewielki i pod kontrolą). Należy być go świadomym i poddawać go kontroli.

Dług technologiczny

	Widoczna	Niewidoczna
Pozytywna wartość	Nowa pożądana właściwość	Architektura
Negatywna wartość	Błąd	Dług technologiczny

Powiązania pomiędzy modułami oprogramowania

- **Code coupling - zależność oprogramowania**. Stopień powiązania danego modułu z innymi. Reprezentuje jego „samodzielność” w realizowaniu zadań. Należy minimalizować zależność oprogramowania.

- **Code cohesion - spójność oprogramowania**. Kod charakteryzujący się wysoką spójnością, ma podobne funkcjonalnie moduły, łatwo testowalne oraz łatwe do zrozumienia. W wysokim stopniu współpracuje z elementami wewnętrznymi danego modułu w celu realizacji celu nadrzędnego. Należy dążyć do jak największej spójności oprogramowania.

Typy spójności oprogramowania

- **Przypadkowa spójność** - elementy są zgrupowane razem ze względu na … łatwość wywołania. Są to wszelkiego rodzaju moduły-narzędzia. Mają całe spektrum niepowiązanych funkcjonalnie narzędzi. Tego rodzaju spójności unikaj.

- **Logiczna spójność** - obsługa podobnych logicznych elementów. Np. obsługa szyn komunikacyjnych różnego rodzaju, urządzeń wprowadzania danych (klawiatury, enkodery, panele dotykowe).

- **Tymczasowa spójność** - dany moduł do wykonania/skończenia operacji, wywołuje inny. Po zakończeniu zadania współpraca się kończy.

- **Spójność proceduralna** - zdefiniowane zadanie, wymaga współpracy modułów.

- **Sekwencyjna spójność** - wyjście jednego modułu, staje się wejściem innego.

- **Funkcjonalna spójność** - moduły są połączone bo realizują wspólne jasno zdefiniowane zadanie. Najlepszy rodzaj spójności.

Typy zależności oprogramowania

- **Zależność zawartości** - moduł wykorzystuje kod innego w danej jego wersji/gałęzi. Najgorszy wysoki (wręcz toksyczny) stopień spójności.

- **Zależność wspólna** - moduły zależne od wspólnych globalnych danych. Zmiana globalnych danych przez jeden z modułów bez uwzględnienia wymagań innych powoduje lawinę błędów. W konsekwencji wymagania innych modułów jawnie i niejawnie implementowane są w bieżącym.

- **Zależność komunikacyjna** - moduły są zależne korzystając ze wspólnego protokołu lub szyny komunikacji.

- **Zależność kontroli** - moduł publikuje informację na temat swojego stanu, innym zainteresowanym modułom.

- **Zależność znaczników** - moduł włącza część struktury danych innego modułu modyfikując ją poza kontrolą modułu podstawowego. Dane w każdym z modułów pełnią inną rolę i tylko ich fragment jest interesujący dla każdego z modułów.

- **Zależność danych** - moduły współdzielą rodzaj elementarnych danych (np. POD). Przesyłają je między sobą i poddają obróbce.

Pożądana sytuacja…

Wysoka spójność najczęściej oznacza niską zależność

Zen programowania...

- KISS - Keep it simple, stupid
- DRY - Don't repeat yourself
- YAGNI - You aren't gonna need it
- Worse is better
- ...

S.O.L.I.D.

- Reguły tworzenia aplikacji S.O.L.I.D.:
    - SRP - Single Responsibility Principle
    - OCP - Open/Closed Principle
    - LSP - Liskov Substitution Principle
    - ISP - Interface Segregation Principle
    - DIP - Dependency Inversion Principle

S.O.L.I.D. - zasady (1/2)

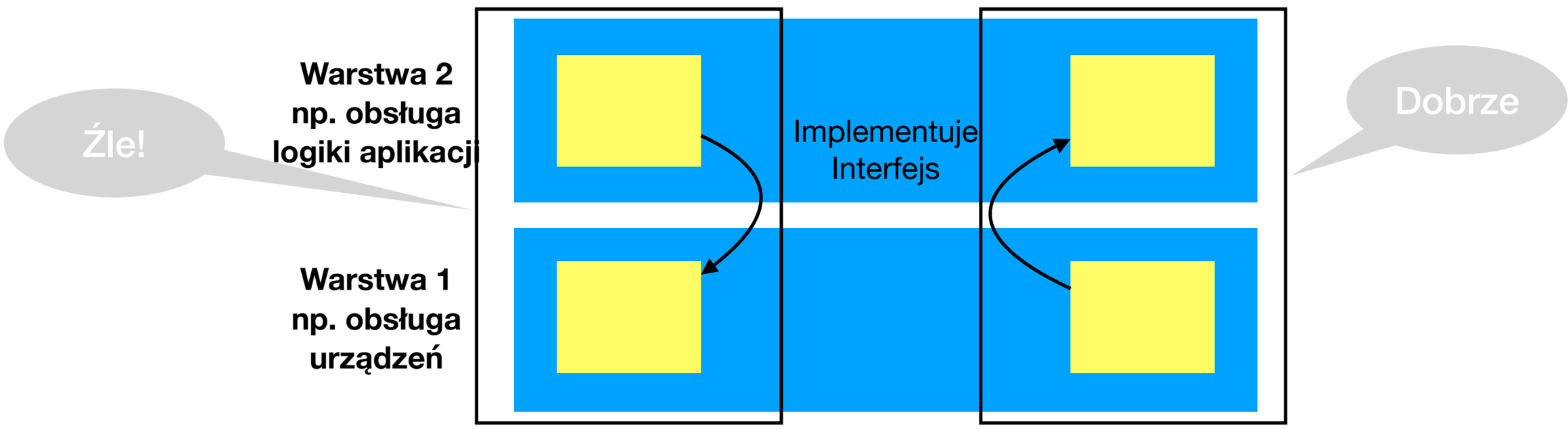
- SRP - komponent jest odpowiedzialny tylko za jedno zadanie. Jest tylko jeden powód do ingerencji w jego implementację. Łamanie tej zasady tworzy zawikłany kod trudny w utrzymaniu.

- OCP - oprogramowanie jest otwarte na dodawanie nowych funkcjonalności i zamknięte na zmiany w kodzie. Dodanie nowych funkcji nie wiąże się z modyfikacją istniejącego kodu ani jego testów.

- LSP - funkcje lub metody akceptujące typy elementów specjalizowanych, mogą akceptować także typy elementów ogólnych. Zalecenia mają wiele wspólnego z DbC oraz dobrymi praktykami obiektowymi.

S.O.L.I.D. - zasady (2/2)

- ISP - twórz interfejsy specjalizowane implementowane w jednym celu. Interfejs nie powinien służyć do zmiany stanu i odczytu statusu modułu. Jego istnienie usprawiedliwione jest realizacją jednego celu. Reguła ma wiele wspólnego z SRP ale dotyczy interfejsów.
- DIP - implementuj interfejsy wyższej warstwy abstrakcji. Dostosowuj się także do sposobu implementacji w wyższej warstwie a nie w warstwie niższej.

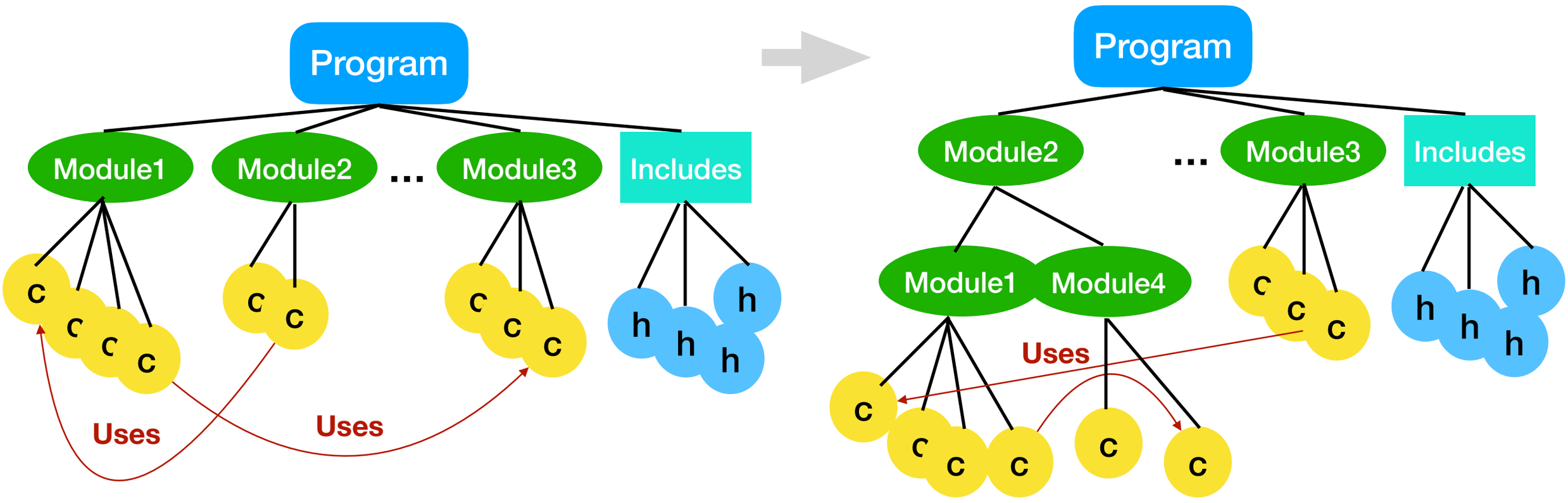


Projekt fizyczny - wstępne rozbicie zależności

- Zależności między plikami projektu, decydują o jego testowaniu, szybkości budowy projektu oraz łatwości wdrożenia personelu.

- Pierwotna struktura projektu powinna być przetworzona tak aby uniknąć zależności cyklicznych oraz zbliżyć się do struktury drzewiastej.

- Procedura nazywana jest poziomowaniem projektu.



Projekt fizyczny - kontrola włączeń nagłówków

- Należy zadbać aby zależności burzące strukturę warstw i drzewa, były wyeliminowane lub zminimalizowane.

- Wskaźnikiem poprawności procedury jest łatwość testowania poszczególnych modułów.

- W przypadku „boskich modułów” (moduł-pająk), może powstać konieczność ich przeprojektowania i dalszej modularyzacji już wyłącznie wynikająca z przesłanek fizycznych.

- Szczególnie rozbudowane nagłówki ze strażnikami, powinny otrzymać włączanie warunkowe (lub *#pragma once*)

O estetyce można dyskutować o skuteczności nie.

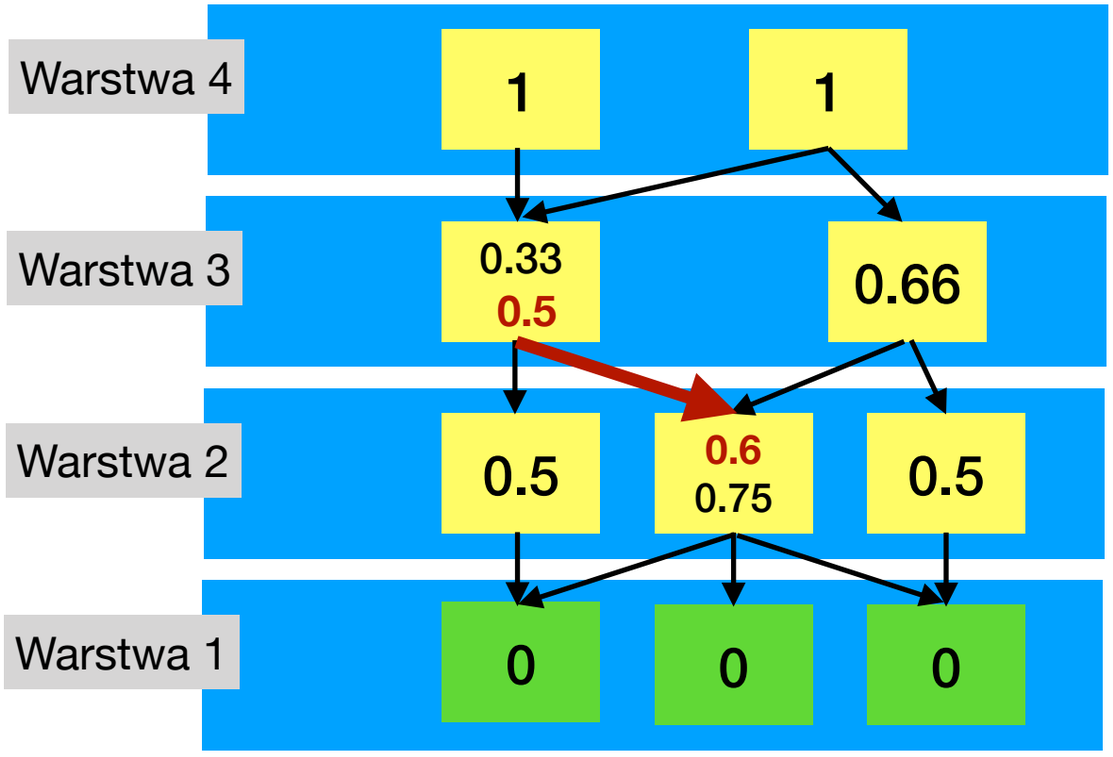
```
#ifndef BIG_H_
#define BIG_H_

#ifndef MOD1_H_
#include ”mod.h”
#endif

#end /* BIG_H_ */
```

Projekt fizyczny - numerowanie warstw

- Należy dążyć do struktury w której można wydzielić warstwy gdzie moduły współpracują wyłącznie z modułami poniżej i nie ma zależności omijających warstwy.
- Zależności cykliczne powinny być wyeliminowane poprzez faktoring.
- **To zrozumiałe że do ideału można nie osiągnąć, należy się do niego zbliżyć jak się da!**

(Nie)Stabilność modułu

- Łatwo identyfikować moduł w którym zmiana pociągnie za **sobą dalekie modyfikacje projektu**.
- W architekturze już wypoziomowanej, wyliczamy współczynnik stabilności.
- 0 - stabilność, 1 - niestabilność modułu



- Ca (ang. *Afferent couplings*) - ilość komponentów zależnych od modułu („wyżej, korzystają z niego”)
- Ce (ang. *Efferent couplings*) - ilość komponentów od których moduł zależy („niżej, korzysta z nich”).
- Niestabilność to:

I = Ce / (Ce + Ca)

(Nie)Stabilność modułu

Im większa niestabilność modułu, tym większy nakład na jego pełne testowanie.

Preferowane zakresy to:

- **I - 0-0.3 i 0.7-1** unikaj wartości pomiędzy
- **Ce - 0-20** - więcej, za wiele pakietów do zrozumienia działania modułu.
- **Ca - 0-500** - więcej, poprawka w module jest krytyczna dla całej aplikacji.

Suma zależności komponentów - CCD

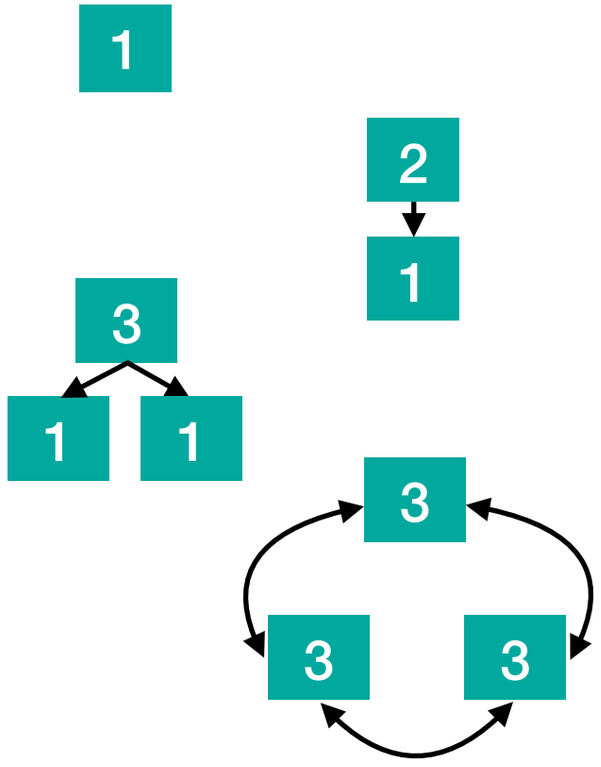
- Jedną z (wielu) miar kontrolujących złożoność architektury, jest CCD (ang. *Cumulative Component Dependency*).

- Wielkość tej miary koreluje z czasem konserwacji i rozwoju programu oraz odzwierciedla koszt jego testowania.

- Moduły programowe opatruje się miarą zależności związanych z ilością niezbędnych do ich przetestowania komponentów.

CCD - zależności

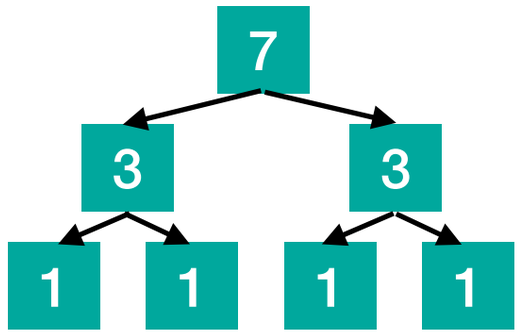
- Sposób opatrywania miarami:
  - Komponent zależy wyłącznie od siebie:
  - Komponent zależy od podrzędnego:
  - Komponent zależy od 2 podrzędnych:
  - Występują zależności cykliczne:
- CCD będzie sumą tych zależności.



CCD - przypadki architektur

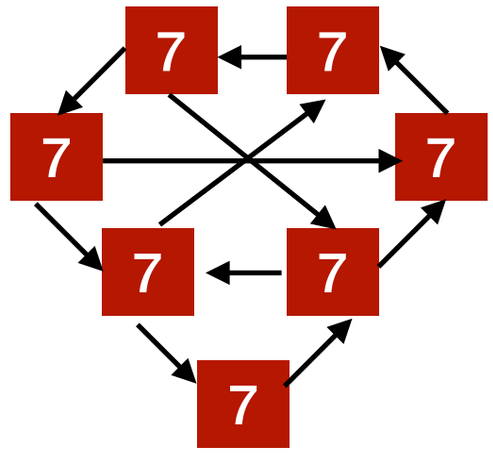
CCD = 28

Rozbudowana aplikacja hierarchiczna



CCD = 17

Zrównoważona aplikacja



CCD = 49

Zależności cykliczne

CCD = 7

Biblioteka lub narzędzia ogólne

CCD - praktycznie

- Nieco więcej informacji można uzyskać dzieląc CCD przez ilość elementów N:

ACCD = CCD / N

- Warto stosować CCD jako wskaźnik „czy jest jeszcze coś do zrobienia” w systemie. Dla architektury pionowej współczynnik wynosi:

CCD = N (N + 1) / 2

- Każda wartość powyżej, świadczy o istniejących zależnościach cyklicznych.

- Warto zdawać sobie sprawę że z jednego współczynnika nie należy tworzyć wyroczni co do jakości pracy zespołu lub architekta.

NCCD

- Więcej informacji co do podobieństwa do modeli wyidealizowanych, uzyskasz odnosząc CCD do wartości CCD dla takiej samej ilości elementów struktury drzewiastej.

- Dla struktury drzewiastej zachodzi następująca zależność:

$$\mathbf{CCD_{(tree)}\ (N) = (N + 1) * (log_2(N + 1) - 1) + 1}$$

- Znormalizowane CCD czyli NCDD, będzie wtedy:

  - Zbliżone do 1 dla architektur podobnych do drzewa.

  - Mniejsze niż 1.0 dla architektur zbliżonych do bibliotek.

  - Mocno ponad 1.0 dla architektur pionowych.

$$\mathbf{NCCD = CCD / CCD_{(tree)}}$$

- Architektury pionowe jednak promują ponowne użycie komponentów stąd NCCD powinno być interpretowane włącznie z CCD!

Narzędzia diagnozy zależności

- Komercyjne: CodeSonar, SonarQube, Code, Squore, CodeClimate…
- Narzędzia na wolnych licencjach:
  - gcc z przełącznikami -M, -MM, -H .. :)
  - cloc - http://cloc.sourceforge.net/ statystyki kodu (linie kodu, komentarze itp…)
  - Doxygen - tak… generuje diagram zależności z użyciem graphviz (dot). Opcja HAVE_DOT w konfiguracji
  - cinclude2dot - http://flourish.org/cinclude2dot/ generuje diagram zależności.
  - cppdeps - http://pastebin.com/raw.php?i=0kMQQKGb oblicza NCCD i poziomuje architekturę włączeń.
  - include-what-you-use (iwyu) - https://github.com/include-what-you-use/include-what-you-use identyfikuje brakujące (a niezbędne) włączenia oraz usuwa zbędne i deklaruje struktury z usunięciem zbędnych nagłówków.

Rozbijanie zależności i faktoring

- Katalog technik (otwarty) usuwania sprzężeń obejmuje dla C:
  - Stosowanie wywołań zwrotnych (ang. *callbacks*).
  - Wstrzykiwanie zależności (ang. *dependency injection*).
  - Stosowanie technik wyniesienia, obniżania, głupich danych (ang. *dump data*).
  - Stosowanie wzorców projektowych.
  - …
- Wybrane techniki omówi instruktor.