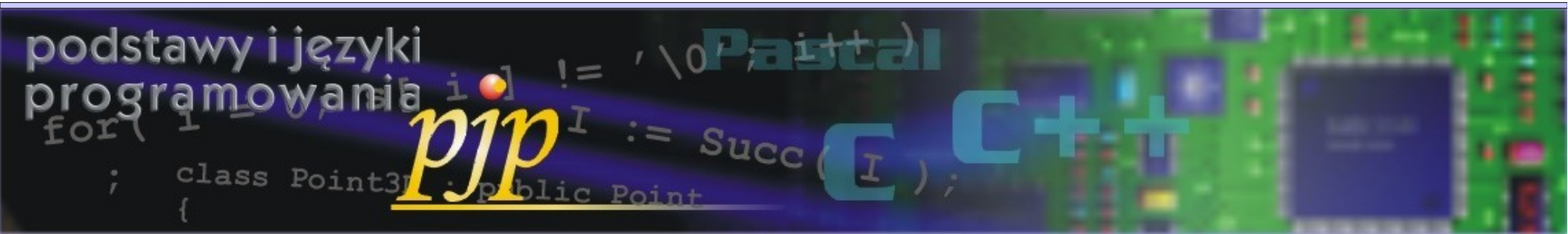
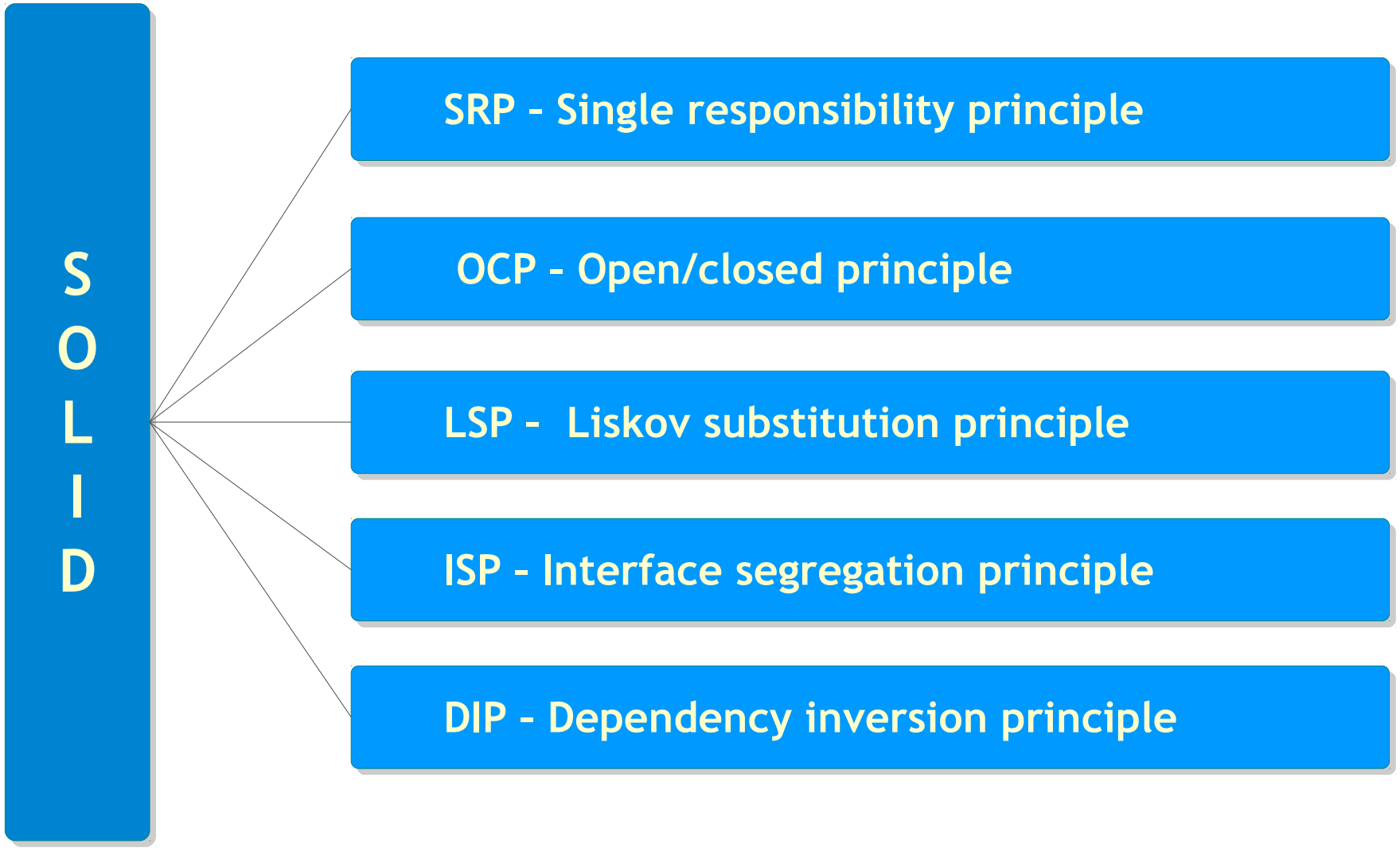
Projektowanie i programowanie obiektowe

Roman Simiński
roman.siminski@us.edu.pl
roman@siminskionline.pl
programowanie.siminskionline.pl

SOLID, KISS, DRY, YAGNI...
Zasady wytwarzania solidnego programowania



SOLID – zbiór potrzebnych zasad poprawiających jakość kodu



SOLID - w poszukiwaniu świętego Graal'a

- Niewątpliwie trzeba dążyć do tego aby kod spełniał reguły SOLID.
- Jednak dążenie do spełnienia wszystkich zasad w realnym projekcie może być trudne, czasem niemożliwe, czasem nieopłacalne.
- Dążenie do spełnienia zasad SOLID często łączone jest i inną zasadą:

DRY - Don’t Repeat Yourself (Nie powtarzaj się)

- DRY – piszemy kod, który nie powtarza logiki w różnych miejscach aplikacji, to jest OK, zgodne SRP.
- Jednak stosowanie DRY dla kodu wielu projektów jest siłowym poszukiwaniem kody „na wszystkie okazje”, znane, przewidywalne i nieprzewidywalne.
- Powoduje to często utworzenie skomplikowanego kodu, tak uniwersalnego i sparametryzowanego, że jest niezrozumiały i zawiły w stosowaniu.
- Poszukiwanie Świętego Graal'a SOLID i DRY często jest narusza zasadę **KISS**.

KISS - uniwersalna zasada projektowania i budowania wszystkiego

KISS - Keep It Simple Stupid (Nie kombinuj głuptasie)

BUZI - Bez Udziwnień Zapisu Idioto

- Piszemy kod tak prosto jak to tylko możliwe, unikając zbędnych udziwnień, priorytetem jest *czytelność* i *zrozumiałość*.
- KISS bywa łączony z **YAGNI**.

YAGNI - pragmatyczne podejście do programowania

YAGNI - You Ain't Gonna Need It (Nie będziesz tego potrzebował)

- YAGNI mówi, że nie należy pisać kodu, który nie jest aktualnie potrzebne, ale wydaje się, że może się kiedyś przydać.
- Możemy marnować czas na coś, co się nie przyda, albo nasza antycypacja może wyć nietrafiona ze względu na nieprzewidywalne zmiany, jakie mogą się pojawić.

YAGNI wywodzi się z obserwacji:

- Przepowiadanie przyszłości (przyszłych zmian) nie jest mocną cechą ludzi, a w tym programistów.
- Koszty poniesione na implementacje przewidywanych zmian mogą być zmarnowane, ponieważ zmiany mogą być całkiem inne.
- Każde elastyczne rozwiązanie staje się skomplikowane.
- Każde elastyczne rozwiązanie rzadko będzie dostatecznie elastyczne.

SILID, DRY, KISS, YAGNI... i kogo stu słuchać?

- Mądrych zasad mających pomóc w wytwarzaniu oprogramowania o dobrej jakości jest więcej: *Tell Don't Ask*, *Separation of Concerns*, *Dependency Injection*, *Inversion of Control*...
- Niektóre zasady mogą pozornie stać wobec siebie w sprzeczności:

YAGNI

Pisz dla aktualnych wymagań, nie spekuluj o przyszłych

DRY

Pisz bez powtórzeń, dbaj o ponowne wykorzystanie kodu, a więc przewiduj przyszłe zmiany i zastosowania

Zmiany specyfikacji - zmora projektanta i programisty

- YAGNI: pozornie zbędne rozbudowanie *czegoś* jest usprawiedliwione jedynie wtedy, gdy koszty wprowadzenia owej rozbudowy w przyszłości będą zdecydowanie wyższe, niż obecnie.
- YAGNI: problemy należy rozwiązywać w miarę ich pojawiania się, co umożliwi skupienie się na zadaniach, które są obecnie istotne i pomoże uniknąć pracy, która może okazać się w przyszłości całkiem niepotrzebna.
- **Czy to nie jest krótkowzroczność?**
- **Czy to czasem nie prowadzi do prostackiego i prymitywnego kodu niepodatnego na zmiany?**

Nie! Bo YAGNI wywodzi się z Extreme Programming, zaliczanej do metodyk zwinnych. Zatem YAGNI stosowane w metodyce XP reguluje taktykę programowania, zarządzaniem zmianami zajmuje się metodyka.

Prosty i spełniający aktualne wymagania kod można relatywnie łatwo modyfikować wykorzystując refaktoryzację, kontrolować poprawność testami jednostkowymi (podejście TDD).

Wracamy do SOLID

SRP — zasada pojedynczej odpowiedzialności

SRP - Single responsibility principle

- W oryginale: *A class should have only one reason to change*.
- Rozwinięcie: *Klasa powinna odpowiadać za jedną rzecz. Zatem powód, żeby ją zmienić, powinien być tyko jeden*.

. . .

OCP — *otwarte* na rozbudowę, *zamknięte* na modyfikacje

OCP - Open/closed principle

- W oryginale: *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*.
- Rozwinięcie: *Komponenty programu powinny być tak skonstruowane, aby zmiany uwzględniane były poprzez rozszerzenie istniejącego kodu a nie jego modyfikację (przeróbkę)*.
- Istotna przyczyna: Chęć uniknięcia współczesnej wersji efektu „*spaghetti code*”.
- Modyfikacja interfejsu pewnej klasy może spowodować błędy we fragmentach kodu wykorzystujących interfejs klasy.
- Kierowanie się zasadą OCP jest szczególnie ważna przy realizacji projektów zespołowych oraz realizacji bibliotek programistycznych i wtyczek programowych.

LSP — zasada podstawienia (Barbary) Liskov

LSP - Liskov substitution principle

- W oryginale: *If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*
- Upraszczając: *In class hierarchies, it should be possible to treat a specialized object as if it were a base class object.*
- Rozwinięcie: *Fragmenty kodu (zwykle funkcje), które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas pochodnych, bez dokładnej znajomości tych obiektów.*
- Internet jest nafaszerowany dziwnymi przykładami ilustrującymi LSP. Niektóre są głupie, niektóre nieadekwatne.
- Złamanie LSP zajdzie wtedy, gdy kod wykorzystujący wskaźnik lub referencję do pewnej klasy bazowej, do swojego poprawnego działania musi identyfikować typ obiektu.

ISP — zasada segmentacji interfejsów

ISP - Interface segregation principle

- W oryginale: *Many client-specific interfaces are better than one general-purpose interface.*
- Rozwinięcie: *Lepiej stosować więcej specyficznych interfejsów niż jeden uniwersalny.*
- Objawem naruszenia ISP jest sytuacja w której nie implementujemy pewnych metod interfejsu. Narażamy się na błędy kompilacji, niechciane wyjątki lub wymuszone stosowanie pustych metod implementujących.
- Skoro nie implementujemy wszystkich metod interfejsu, to pewnie są one w danym kontekście niepotrzebne.
- Warto zatem przemyśleć konstrukcję takiego interfejsu i podzielić go na pewną liczbę interfejsów prostszych, tematycznie spójnych.

DIP — zasada odwrócenia zależności

DIP - Dependency inversion principle

- W oryginale: *Many client-specific interfaces are better than one general-purpose interface.*
- Rozwinięcie: *Lepiej stosować więcej specyficznych interfejsów niż jeden uniwersalny.*
- Objawem naruszenia ISP jest sytuacja w której klasa nadrzędna zależy od klasy podrzędnej. Wystąpi to wtedy, gdy klasa zawiera zakodowaną stałą zależność od konkretnej klasy.
- Próba podmiany klasy prowadzi do konieczności modyfikacji kodu.
- Odwrócenie zależności realizować można poprzez wykorzystanie *wzorca* IoC (*Inversion of Control*) oraz DI (*Dependency Injection*).

Czas na konkretne Przykłady...

Ale czy naprawdę rozumiemy dziedziczenie, klasy abstrakcyjne, polimorfizm, metody wirtualne, interfejsy, (wskaźniki/referencje), ...?