

*NAJLEPSZE STRATEGIE
SKUTECZNYCH
PROGRAMISTÓW. TECHNIKI
PRACY Z KODEM*

KOD: NSKOD

OPIS

Praca programisty oprócz umiejętności i wiedzy technicznej, wymaga również doskonałej pracy z kodem. Umiejętności te są często marginalizowane, gdyż liczy się ostateczny efekt – działający system. Jednak tworzenie oprogramowania to bardzo złożony proces i elementarna jakość stworzonego kodu jest kluczowa podczas rozwijania systemów. Szkolenie to dotyczy technik tworzenia czytelnego, łatwego do utrzymania kodu w oparciu o metody obiektowe.

PROFIL UCZESTNIKA

Programista:

- chce świadomie i efektywnie tworzyć kod;
- chce świadomie i efektywnie utrzymywać kod;
- chce optymalnie wykorzystywać możliwości udostępniane przez współczesne języki programowania.

KORZYŚCI ZE SZKOLENIA

1. **Programiści optymalnie wykorzystują czas poświęcony na pracę** – ponieważ używają efektywnych metod tworzenia kodu.
2. **Czas poświęcony na rozwiązywanie problemów ulega skróceniu** – programiści, dzięki zwiększonej samoświadomości w tworzeniu kodu, potrafią wykształcać u siebie nowe nawyki i sposoby postępowania poprawiające efektywność pracy.
3. **Programiści mają większą motywację do pracy** – gdyż używanie technik daje poczucie kontroli nad tworzonymi rozwiązaniami, a w konsekwencji większe zadowolenie w wykonywanej pracy.
4. **Zmniejsza się koszt realizacji projektów** – dzięki zwiększeniu efektywności pracy programistów.
5. **Zmniejsza się koszt utrzymywania i rozwijania aplikacji** – ponieważ programiści tworzą oprogramowanie wysokiej jakości.

PARAMETRY SZKOLENIA

Czas trwania: 3 dni – 24 godzin.

Forma zajęć: Laboratorium Efektywności Programisty - 60%, wykład – 40%.

Wielkość grupy: do 10 osób.

SZCZEGÓŁOWY PROGRAM

Moduły szkoleniowe	Nabyte wiedza i umiejętności, poruszane zagadnienia
Software Cratsmanship	<ul style="list-style-type: none">• Idea Software Craftsmanship• Koszt kiepskiego kodu• Efektywność pracy z kodem
Programowanie obiektowe	<ul style="list-style-type: none">• Kiedy kod jest naprawdę obiektowy• Zasada SOLID• Kompozycja a dziedziczenie• Odpowiedzialność klas, metod, pakietów, modułów• Czego można się nauczyć z wzorców projektowych• Inversion of Control oraz Dependency Injection• Modele architektoniczne• Architektury wielowarstwowe
Klasy i ich stan	<ul style="list-style-type: none">• Poprawnie zdefiniowana klasa• Hermetyzacja i izolowanie zmian• Nazywanie klas i metod• Kiedy używać interfejsów i klas abstrakcyjnych• Wzorce implementacyjne: Value Object, Implementator, Contitional, Delegation, Pluggable Selector, Library Class

	<ul style="list-style-type: none"> • Przechowywanie stanu • Wzorce implementacyjne: Direct Access, Indirect Access, Collecting Parameter, Optional Parameter, Eager Initialization • Antysymetria danych i obiektów • Prawo Demeter • Data Transfer Object
Czytelność kodu	<ul style="list-style-type: none"> • Reguły tworzenia czytelnych nazw • Nazwy a dziedzina problemu • Nazwy a kontekst • Małe funkcje • Jeden poziom abstrakcji • Argumenty funkcji i zwracany typ a nazwa funkcji • Efekty uboczne • Zapytania i polecenia • Don't Repeat Yourself • Czy komentarze są potrzebne? • Zen tworzenia komentarzy • Komentarze a refaktoryzacja
Obsługa sytuacji wyjątkowych	<ul style="list-style-type: none"> • Wyjątki a wyniki zwracane • Kontrolowane czy niekontrolowane wyjątki? • Tworzenie hierarchii wyjątków • Obsługa referencji null • Dostarczanie kontekstu • Praktyki tworzenia konstrukcji try-catch-finally
TDD	<ul style="list-style-type: none"> • Test-Driven Development - rewolucja w programowaniu • W TDD nie chodzi o testowanie • Red-green-refactor czyli TDD w 15 minut

	<ul style="list-style-type: none">• Zasady tworzenia testów jednostkowych• Jak utrzymywać testy?
Refaktoryzacja	<ul style="list-style-type: none">• Idea refaktoryzacji• Cztery podstawowe techniki refaktoryzacji• Technika dekompozycji algorytmu• Małe kroki• Kiedy refaktoryzacja jest opłacalna?• Refaktoryzacja w projektach odziedziczonych