Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Inżynieria Oprogramowania dla Elektromobilności

Dodatek. Dobre nawyki w programowaniu

Formatowanie kodu

Należy stosować spójny zestaw reguł dotyczących formatowania kodu w celu poprawy jego czytelności. Do podstawowych zasad można zaliczyć te odnoszące się do: tworzenia wcięć w kodzie; stosowanej konwencji notacji; sposobów ograniczania bloków kodu; maksymalnej długości linii kodu; dokumentacji kodu w postaci komentarzy. Formatowanie kodu nie ma wpływu na proces kompilacji, jednakże w sposób znaczący ułatwia pracę nad rozwojem oprogramowania, prowadząc do obniżenia nakładów finansowych i czasu implementacji.

Konwencje notacji

Wyróżnia się wiele konwencji dotyczących nazewnictwa zmiennych, stałych i funkcji. Do najczęściej stosowanych należą m.in. **camelCase**, **PascalCase** oraz **snake_case**. Wybór konwencji notacji jest najczęściej kwestią upodobania. Istotnym jest, aby powzięta konwencja notacji była spójna, tj. nie należy łączyć różnych konwencji w obrębie jednego projektu aplikacji. Nadawane nazwy zmiennych i funkcji powinne być wymowne, aby na ich podstawie można było wywnioskować ich przeznaczenie. Niepisaną regułą stało się nadawanie nazw w języku angielskim z powodu jego powszechności i spójności językowej ze słowami kluczowymi kompilatora.

Dokumentacja

Równocześnie z modyfikowanym kodem powinno aktualizować się jego dokumentację. Warto umieszczać komentarze w plikach interfejsów oraz w implementacji złożonych algorytmów. Komentarze w obrębie poszczególnych plików (głównie nagłówkowych i źródłowych) mogą służyć oznaczaniu praw autorskich oraz warunków udzielenia licencji.

Magiczne liczby

Magiczną liczbą nazywa się wartość liczbową użytą pojedynczo lub wielokrotnie w kodzie w sposób bezpośredni bez objaśnienia jej znaczenia. Nienazwane zmienne liczbowe nieopatrzone komentarzem uznawane są za antywzorzec, ponieważ znacznie zmniejszają czytelność kodu. Dobrym zwyczajem jest deklarowanie wszystkich wartości liczbowych jako stałych opatrzonych jednoznaczną nazwą.

Przedwczesna optymalizacja

Przedwczesna optymalizacja (*ang. premature optimization*) polega na przeprowadzaniu procesu optymalizacji kodu na małą skalę w początkowych etapach rozwoju oprogramowania i jest uznawana za niewłaściwą praktykę. We wczesnych fazach rozwoju oprogramowania należy się skupić na wytworzeniu czytelnego, zrozumiałego, łatwego w utrzymaniu i w miarę możliwości zwięzłego kodu. Optymalizacja powinna odbywać się dopiero po przeprowadzeniu analizy wydajności systemu i identyfikacji tzw. ***wąskich gardeł*** (*ang. bottlenecks*), czyli fragmentów kodu o krytycznym znaczeniu dla szybkości działania i zużycia zasobów systemowych. ***Wąskie gardła*** stanowią z reguły jedynie kilka procent całości kodu aplikacji, w związku z czym w praktyce potrzebna jest jedynie optymalizacja na skalę lokalną. ***Przedwczesna optymalizacja*** prowadzi do znacznego wydłużenia czasu implementacji i testowania oprogramowania; może obniżać czytelność kodu, zwiększać jego złożoność cyklomatyczną, a także skutkować wprowadzeniem nowych błędów oprogramowania.

Agregacja a dziedziczenie

Należy stosować dziedziczenie, jeśli klasa pochodna wykorzystuje prawie wszystkie metody interfejsu klasy bazowej. W innych przypadkach agregacja obiektów powinna być preferowana zamiast dziedziczenia. Należy unikać głębokich drzew dziedziczenia.

Luźne powiązania między komponentami

Stosowanie ***luźnych powiązań*** (*ang. loose coupling*) między komponentami jest preferowane. Luźno powiązane komponenty mogą być wymieniane i implementowane niezależnie od siebie. ***Luźne powiązania*** mogą być realizowane przez wprowadzanie abstrakcyjnych interfejsów; rozdzielanie abstrakcji (interfejsu) od implementacji (wzorzec projektowy ***Mostu*** (*ang. Bridge*)); stosowanie wzorca ***Wstrzykiwania zależności*** (*ang. Dependency injection*)); preferowanie agregacji i asocjacji zamiast kompozycji obiektów.

Singletony

Należy unikać ***singletonów***, które uznawane są za antywzorce projektowe ze względu na: utrudnione testowanie kodu; wprowadzanie ukrytych zależności przez globalny dostęp do obiektu; problematyczne niszczenie instancji obiektu (odpowiedzialność za tworzenie obiektu wewnątrz klasy).

Testy oprogramowania

Każda pojedyncza funkcjonalność w obrębie metody (funkcji) powinna mieć dedytkowany test jednostkowy. Interakcja między komponentami i podsystemami powinna być weryfikowana za pomocą testów integracyjnych. Automatyzacja procesu testowania stanowi uzupełnienie, a nie alternatywę dla testów manualnych.

Reguła YAGNI (*You aren't gonna need it*)

Nie należy tworzyć kodu, który nie jest aktualnie potrzebny. W przeciwnym razie napisany kod może wymagać refaktoryzacji w wyniku zmiany wymagań lub nigdy nie zostać wykonany, stając się tzw. ***martwym kodem*** (*ang. dead code*), jednocześnie zwiększając rozmiar pliku wynikowego.

Reguła DRY (*Don't repeat yourself*)

W procesie tworzenia oprogramowania należy unikać powtórzeń. Dotyczy to zarówno duplikowania kodu, jak i wykonywania powtarzających się czynności w procesie budowania pliku wykonywalnego aplikacji. Należy wydzielać powtarzające się fragmenty kodu do osobnych funkcji, makr lub szablonów oraz, w miarę możliwości, automatyzować proces kompilacji i konsolidacji. Zyskiem ze stosowania ***reguły DRY*** są: oszczędność czasu, możliwość ponownego wykorzystania kodu (*ang. code reuse*), przenośność kodu, zmniejszenie prawdopodobieństwa wprowadzenia nowych błędów oprogramowania.

Zasady SOLID

Single responsibility principle (*Zasada pojedynczej odpowiedzialności*)
Open/closed principle (*Zasada otwarte–zamknięte*)
Liskov substitution principle (*Zasada podstawienia Liskov*)
Interface segregation principle (*Zasada segregacji interfejsów*)
Dependency inversion principle (*Zasada odwrócenia zależności*)

Zasada pojedynczej odpowiedzialności

Nie powinien istnieć więcej niż jeden powód do modyfikacji klasy (funkcji). Klasa (funkcja) powinna mieć tylko jedną odpowiedzialność (unikanie antywzorca ***Boskiej klasy*** (*ang. God class*)). Kod należy dzielić na jednostki translacji ze względu na funkcjonalność.

Zasada otwarte–zamknięte

Kod powinien być otwarty na rozszerzenia i zamknięty na modyfikacje.

Zasada podstawienia Liskov

Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również instancji klas pochodnych, bez dokładnej znajomości tych obiektów.

Zasada segregacji interfejsów

Interfejs nie powinien zawierać metod, których nie używa implementująca go klasa. Rozbudowane interfejsy powinny być dzielone na mniejsze, dedykowane interfejsy.

Zasada odwrócenia zależności

Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych. Interfejsy nie powinny zależeć od implementacji. Relacje w drzewie dziedziczenia powinny w sposób naturalny wynikać z abstrakcji.