Program szkolenia:

Praca z kodem „legacy”: strategie, naprawa błędów, refaktoryzacja oraz narzędzia wspomagające.

Informacje:

Nazwa:	**Praca z kodem „legacy”: strategie, naprawa błędów, refaktoryzacja oraz narzędzia wspomagające.**
Kod:	**ccpp-legacy C++**
Kategoria:	C i C++
Odbiorcy:	developerzy
Czas trwania:	3 dni
Forma:	40% wykłady / 60% warsztaty

Szkolenie prezentuje pragmatyczne podejście do pracy z kodem „legacy”. Przedstawione strategie i narzędzia pozwalają przełamać strach przed zmianami w zastanym kodzie, przez co ułatwiają bezpieczne modyfikacje oraz refaktoryzację kodu kładąc szczególny nacisk na testowanie.

W oparciu o rzeczywiste przykłady, uczestnicy zdobywają wiedzę i praktykę w metodycznym podejściu do zmian w zaniedbanym kodzie.

Szkolenie przeznaczone jest dla programistów C++ rozwijających lub utrztmujących kod w (najczęściej) odziedziczonych projektach. Zdobyta wiedza przekłada się w praktyczny sposób na jakość dostarczanych zmian mierzoną w szerszej perspektywie czasu.

Zalety szkolenia:

- Szkolenie prowadzone przez trenera ze dużym doświadczeniem w pracy z kodem legacy
- Konstrukcja techniczna szkolenia umożliwia nabycie praktycznych umiejętności wprowadzania szeroko rozumianych zmian w kodzie legacy
- Rzeczywiste przykłady

BOTTEGA IT minds

Szczegółowy program:

1. „Legacy Code” – czy to tylko kod odziedziczony?

1.1. Definicja LC – "oddzielenie ziarna od plew”

1.1.1. „Wiedza plemienna” – znajomość funkcjonalności i wymiana wiedzy w projekcie

1.1.2. Kod niepokryty testami

1.1.3. „Zapachy kodu” – zbiór najczęstszych błędów/praktyk w zaniedbanym kodzie

1.1.3.1. Długa metoda – brak kohezji

1.1.3.2. Powielanie kodu – „nie ma to jak powielać te same błędy”

1.1.3.3. Eksplozja kombinatoryczna – „siostra powielania, czyli robię to samo na innych danych”

1.1.3.4. Osobliwe rozwiązanie – „cichy brat powielania, czyli zrobię to samo ale w nieco inny sposób”

1.1.3.5. Złożoność warunków – „bo trzeba napisać tak by nikt inny tego nie był w stanie zrozumieć”

1.1.3.6. Rozrzucanie rozwiązania – strach przed zmianami w kilku miejscach

1.1.3.7. Klasa bóg – „klasa może i robi wszystko”

1.1.3.8. Leniwa klasa – „klasa nie zarabia na siebie”

1.1.3.9. Obnażanie się – „wszystko w public”

1.2. Rodzaje zmian w LC ze szczególnym ujęciem kalkulacji ryzyka na rzecz świadomego dobrania rozwiązania do klasy problemu

1.2.1. Przyczyny wprowadzania zmian

1.2.1.1. Naprawa błędów

1.2.1.2. Rozszerzanie funkcjonalności

1.2.1.3. Refaktoryzacja

1.2.2. Kalkulacja i minimalizacja ryzyka

1.2.2.1. Jasne określenie punktu widzenia klienta

BO·TT·EGA
IT minds

1.2.2.2. Ocena klasy problemu i stopnia komplikacji wprowadzanej zmiany

1.2.2.3. Stopień opłacalności różnych rozwiązań w świetle testów

1.2.2.4. Regresja – wpływ dokonanej (często jednostkowej) zmiany na szereg pozostałych funkcjonalności oraz na lokalną i globalną architekturę

1.2.2.5. Nowe testy – koszt napisania i utrzymania vs. posiadanie i bezpieczeństwo kolejnych modyfikacji

2. Metodyki wprowadzania zmian – „jak to zrobić by nie napsuć”

2.1. Naprawa błędów ze szczególnym uwzględnieniem wskaźników i tablic

2.1.1. Podejście naiwne

2.1.2. Podejście „imperatywne” – wykorzystanie mechanizmów języka i kompilacji na rzecz bezpieczeństwa i jakości wprowadzonych poprawek (mała refaktoryzacja).

2.2. Rozszerzanie funkcjonalności

2.2.1. Podejście naiwne

2.2.2. Rozszerzania funkcjonalności w oparciu o refaktoryzację

3. Wstęp do refaktoryzacji

3.1. Czym jest dobra refaktoryzacja?

3.2. Kiedy (nie)można lub (nie)należy dokonać refaktoryzacji?

3.3. Narzędzia wspomagające

3.3.1. Cppcheck – sprawdzenie poprawności kodu bez uruchamiania go

3.3.2. CPD – wykrywanie zduplikowanego kodu (wstęp do refaktoryzacji właściwej)

3.3.3. CCCC – „lokalny” stopień złożoności kodu

3.3.4. Valgrind – analiza sterty (memcheck), śledzenie wskaźników (ptrcheck), sprawdzenie wielowątkowości (hellgrind)

3.4. Strategia – testowanie punktem wyjścia na rzecz tworzenia bezpiecznych wysp i późniejszego scalenia ich w jeden pokryty testami kontynent

4. TDR – Test Driven Refactoring

4.1. Identyfikacji newralgicznych punktów kluczem do sukcesu

4.2. Pokrycie testami newralgicznych punktów

4.2.1. Łamanie zależności zewnętrznych i lokalnych

4.2.2. Pisanie testów i wprowadzania mock'ów z uchwyceniem najczęściej popełnianych błędów

4.3. Skutki cząstkowego podejścia

5. Refaktoryzacja właściwa – budowa mikro/makro architektury w oparciu o wzorce

5.1. Upraszczanie kodu

5.1.1. Ekstrakcja metod

5.1.2. Zastępowanie wyrażeń warunkowych wzorcem strategii

5.1.3. Zmiana upiększeń w dekorator

5.1.4. Zastępowanie wyrażeń zmian stanu klasami stanu

5.2. Tworzenie obiektów

5.2.1. Metody tworzące egzemplarze zamiast naiwnych konstruktorów

5.2.2. Operacja tworzenia obiektów w fabryce

5.2.3. Klasy builder do hermetyzacji obiektów kompozytowych

5.2.4. Inline singleton – a może zło wcielone

5.3. Uogólnianie kodu

5.3.1. Metoda szablonowa – wprowadź dziedziczenie na rzecz polimorfizmu

5.3.2. Kompozyt – zamiana relacji jeden do wielu

5.3.3. Obserwator dla notyfikacji

5.3.4. Adapter – unifikacja interfejsów

5.3.5. Interpreter – obsługa niejawnych języków

5.4. Ochrona

5.4.1. Zastępowanie typów prostych klasami „value object"

5.4.2. Akumulacja

5.4.3. Gromadzenie danych w parametrze zbierającym

5.4.4. Gromadzenie danych przy użyciu inspektora (visitor)

6. Wnioskowanie i zabezpieczenia

6.1. Ocena wprowadzonych zmian

6.1.1. Wyniki testów oraz ich pokrycie

6.1.2. Statyczna analiza kodu

6.1.3. Code Review – część procesu dostarczania zmian z szeregiem zalet

6.2. Pętla refaktoryzaji

6.2.1. Czym jest?

6.2.2. Kiedy ją przerwać?

6.3. Testy i ich regresyjność

6.3.1. Reżim wykonywania testów: jednostkowych, modułowych, funkckjonalnych i integracyjnych

6.3.2. Wstęp do CI – „Contineous Integration” – testy ciągłą i integralną częścią produkcji kodu