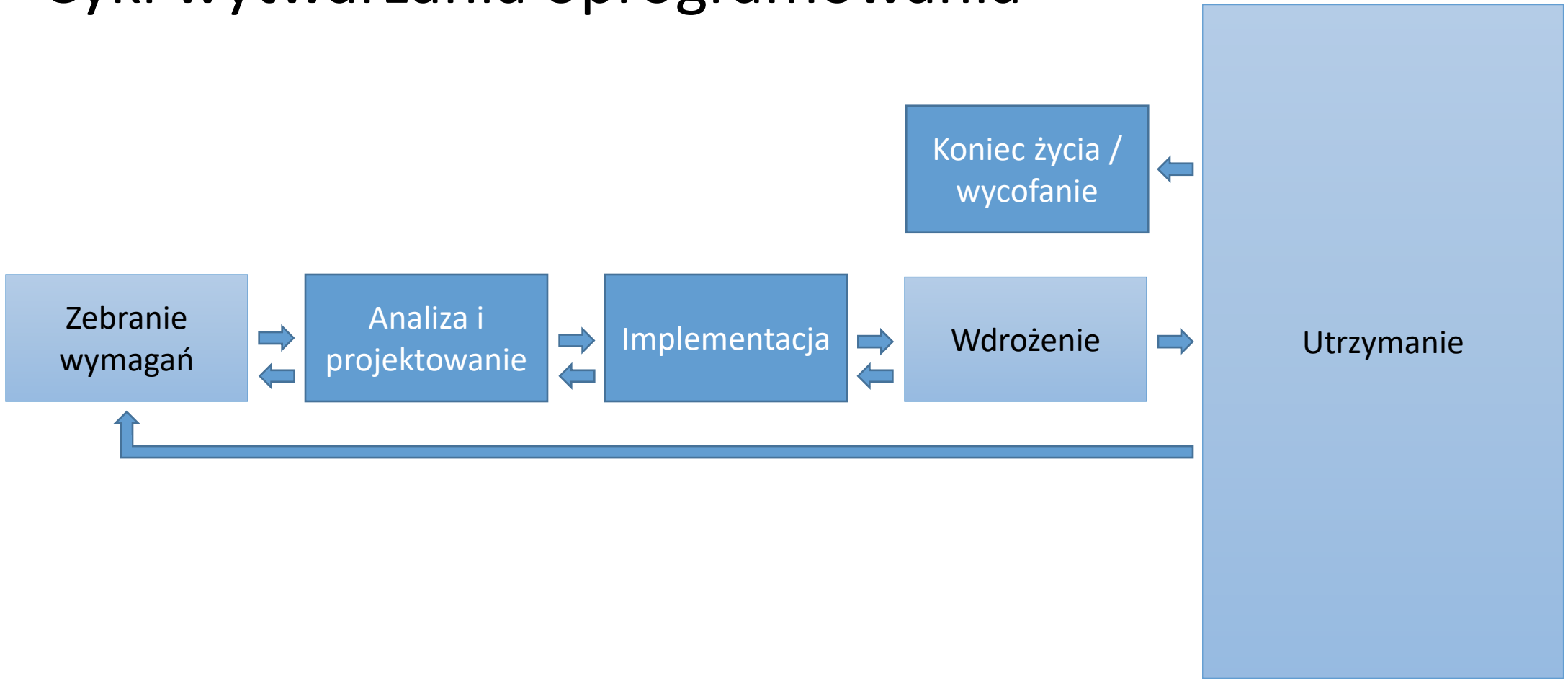
Cykl wytwarzania (życia) oprogramowania c.d.

Sztuka Wytwarzania Oprogramowania, w. 7

Konrad Grochowski

Cykl wytwarzania oprogramowania



Cykl życia - początek

- Od czego zaczyna się życie programu?
- Od wymagań!
  - Nawet jak się nam/komuś wydaje, że nie…

Wymagania

- Sformułowana potrzeba cechy produktu / usługi
  (bądź sposobu działania / realizacji usługi)
- Skąd:
  - Wewnętrznie – analiza rynku, poprzednich wersji, strategia firmy,
  - Zewnętrznie – dostarczone w całości przez klienta (przetargi),
  - Mieszane – ogólna potrzeba klienta jest uszczegóławiana w ramach usługi
- Klasyczny podział:
  - Wymagania funkcjonalne (*co ma robić?*)
  - Wymaganie niefunkcjonalne („wszystko inne”- wydajność, niezawodność, jakość etc., *jak ma robić?*)

Wymagania do wymagań

- Bez dobrych wymagań (dobrze uzgodnionych) *nie da się* zrealizować projektu (w zaplanowanym czasie i budżecie)
- Problem – programiści (twórcy oprogramowania) często nie znają dziedziny (i nie są docelowymi użytkownikami systemów, które tworzą)
- Z drugiej strony – poświęcanie czasu na „dogranie” wymagań może być kosztowne – metodyki *agile* (zwinne) sugerują robić to razem z prototypowaniem / dostarczaniem wersji pośrednich
- Ale nawet „w *agile*” obowiązują cechy „dobrego wymagania”
- Czyli inżynier *musi* umieć ocenić i *wymagać dobrych wymagań*

Dobre wymaganie

- **Jasne** (*Clear*) – zwarty, jasny opis, bez „lania wody” etc.
  - Często sformalizowany/uproszczony język
  - „Ustandaryzowane” konstrukcje gramatyczne (stały schemat zdań)
  - *Koszmarek: Celem biznesowym systemu jest zwiększenie satysfakcji klienta przez udostępnienie mu możliwości zaspokojenia potrzeby wygodnej realizacji obsługi ...*
  - *Lepiej: System ma obsłużyć ...*
- **Kompletne** (*Complete*) – wszystkie informacje potrzebne do zrozumienia (i sprawdzenia) wymagania są w nim zawarte
  - *Źle: System ma być wydajny*
  - *Lepiej: System ma obsłużyć do 1000 zapytań HTTP na sekundę*

Dobre wymaganie

- **Zgodne** (*Consistent*) – nie może być sprzeczne z innymi
  - w zbiorze wymagań nie może być wewnętrznych sprzeczności
  - nie może być też sprzeczności z dokumentami zewnętrznymi
- **Spójne** (*Cohesive*) – w opisie systemu nie powinno być braków
  - ani „nachodzących” na siebie wymagań
  - jedno i drugie otwiera „pole do interpretacji”, czemu wymagania mają zapobiegać
  - dla pojedynczego wymagania:
    Wymaganie odnosi się do jednej i tylko jednej sprawy.

Dobre wymaganie

- **Atomowe** (*Atomic*) – niepodzielne, bo ciężko weryfikować coś, co ma w sobie „i” (a jeszcze gorzej „lub”)
  - *Źle: System będzie wspierał formaty danych: JSON, XML*
  - *Lepiej:*
    System pozwoli na zapis danych do pliku w formacie JSON
    System pozwoli na zapis danych do pliku w formacie XML

- **Jednoznaczne** (*Unambiguous*) – wszystkie pojęcia użyte w definicji wymagania powinny być zrozumiałe dla każdego odbiorcy (skróty i żargon etc. ograniczone do ustalonego – najlepiej sformalizowanego – słownika pojęć)
  - Żargon klienta i programisty może być *drastycznie* różny

Dobre wymaganie

- **Weryfikowalne** (*Verifiable*) – zrealizowanie wymagania powinno dać się *jednoznacznie* i możliwie *obiektywnie* ocenić
  - Ważne i dla klienta i dla producenta (wypłata)
  - Najlepiej, jeśli wymaganie jest ***testowalne***
  - Metoda weryfikacji powinna być zdefiniowana razem z wymaganiem, np.:
    - Test
    - Analiza dokumentacji
    - Niezależna inspekcja

Dobre wymaganie

- **Może mieć priorytet** – w jakiej kolejności / czy musi być zrealizowane
  - *shall/must, should, could*
- **Może mieć uzasadnienie** – przydatne, jeśli to *my* je tworzymy
  - warto jakoś zanotować, skąd dana rzecz się wzięła
- **Może mieć możliwość *śledzenia* (traceability)** – posiada powiązanie z wymaganiami wyższego rzędu / standardami
  - często najlepsza forma uzasadnienia
  - ważne w sformalizowanych procesach (ale nie tylko)

Zbieranie wymagań / analiza wymagań

- Etap rozwoju oprogramowania niekiedy traktowany pobieżnie
  - „są wymagania jakieś to robimy, zobaczymy co wyjdzie”
    (to ani agile, ani extreeme programming)
- Faza kluczowa dla ustalenia celu projektu i zdefiniowania co usatysfakcjonuje klienta
- Dobre wykonanie analizy często wymaga zestawienia *sprawnego* kanału komunikacji od klienta aż do „technicznych”

Cykl życia c.d.

- Mamy wymagania
- Ale jak je weryfikować?
- Warto teraz porozmawiać o elemencie cyklu o kluczowym znaczeniu...

Testowanie oprogramowania

- Z punktu widzenia klienta – najważniejsza część procesu wytwarzania oprogramowania
  - Nawet jeśli klient nie zdaje sobie z tego sprawy...
  - Może warto nazywać to *zapewnianiem jakości*?
- Weryfikuje zgodność programu z oczekiwaniami
- *Nic innego* nie daje informacji, *co* program *naprawdę* robi
  - Istnieją metody weryfikacji formalnej, jednak nie zawsze daje się je zastosować, są drogie, i często mogą być użyte do weryfikacji fragmentów „odseparowanych od świata zewnętrznego”.
  - Modelowanie może pomagać, ale zależy od procesu (czy model jest *przetwarzany* czy *przepisywany*) ale i jakości samego modelu

Testowanie oprogramowania

- Testowanie może tylko zweryfikować,
  czy nie ma ***znanych błędów***
- Testy robi się *na coś* i tak naprawdę *„zielone testy”* to *„negatywne wyniki”*
- Czyli testowanie nie daje 100% gwarancji poprawności działania
- Ale to nie znaczy, że klient ma testować „na produkcji”...

Testowanie oprogramowania

- Rodzajów testów oprogramowania jest bardzo dużo
- W pewnym sensie każde uruchomienie programu i obserwacja jego zachowania to test
  - Tylko najlepiej, żeby klient obserwował tylko oczekiwane działanie...
- Istnieją dedykowani specjaliści zajmujący aspektami testowania
  - Product Assurance / Quality Assurance
    (choć w tych terminach jest więcej, niż tylko testy)
- Testowanie nie musi dotyczyć wyłącznie kodu
  - Np. zgodność instrukcji użytkownika z rzeczywistym oprogramowaniem

Podział testów

- Ze względu na to czy program jest fizycznie uruchamiany:
  - Statyczne
  - Dynamiczne

- Ze względu na obiekty objęte testem (poziom testu):
  - Jednostkowe
  - Integracyjne
  - Systemowe
  - *Akceptacyjne*

- Testy bezpośrednio weryfikujące wymagania czasem nazywa się *testami walidacyjnymi*

Podział testów

- Ze względu na podejście do obserwacji wyniku:
  - White-box
  - Black-box
  - Grey-box
- Ze względu na sposób przeprowadzenia:
  - Automatyczne
  - Manualne

Podział testów

- Ze względu na badane aspekty:
  - Testy funkcjonalne (functional testing)
  - Testy wydajnościowe (performance testing)
  - Testy przeciążeniowe (stress testing)
  - Testy bezpieczeństwa (security testing)
  - Testy bezpieczeństwa (safety testing)
  - Testy niezawodności (dependability testing)
  - Testy odporności (recovery testing)
  - Testy zgodności (compatibility testing)
  - Testy dokumentacji (documentation testing)
  - ...

Inne przykłady rodzajów testów

- Testy dymne (smoke tests)
- Testy regresji (regression testing)
- Testy destrukcyjne (destructive testing)
- Testy „interakcji z człowiekiem”
  - Testy tłumaczeń i regionalizacji (localization and internationalization)
  - Testy używalności (usability)
  - Testy dostępności (accessibility)

Dobry test

- Jednoznacznie ustalony cel testu
- Czytelny scenariusz testu
- Określony warunek końca testu
- Określony warunek oceny testu (zaakceptowania)
- Jeśli możliwe – odwołanie do wymagań, z których test wynika.
- Idealnie:
  - Automatyczny
  - Deterministyczny i powtarzalny
  - Jednoznacznie i łatwo oceniany
  - Szybki

Testowanie jednostkowe

- „Najwyższa stopa zwrotu w jakości”
  - Krótka ścieżka pomiędzy inwestycją/czasem a wynikiem i wpływem na kod
- „Najlepszy przyjaciel programisty”
  - Tylko test mówi, co kod naprawdę robi, więc test jednostkowy pozwala wiedzieć co jest zrobione „na koniec dnia”
- Wymusza lepszą architekturę (decoupling)
- Brak formalnej definicji jednostki
- Nie wszystko da się unit-testować
- Nie zawsze odpowiada bezpośrednio wymaganiom
  - Usunie drobne błędy, ale nie poważne naruszenia funkcjonalności

Dziękuję za uwagę

Konrad.Grochowski@pw.edu.pl