

# Cykl wytwarzania (życia) oprogramowania c.d.

*Sztuka Wytwarzania Oprogramowania, w. 7*

Konrad Grochowski

Instytut Informatyki, Politechnika Warszawska, 2025 ©



# 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

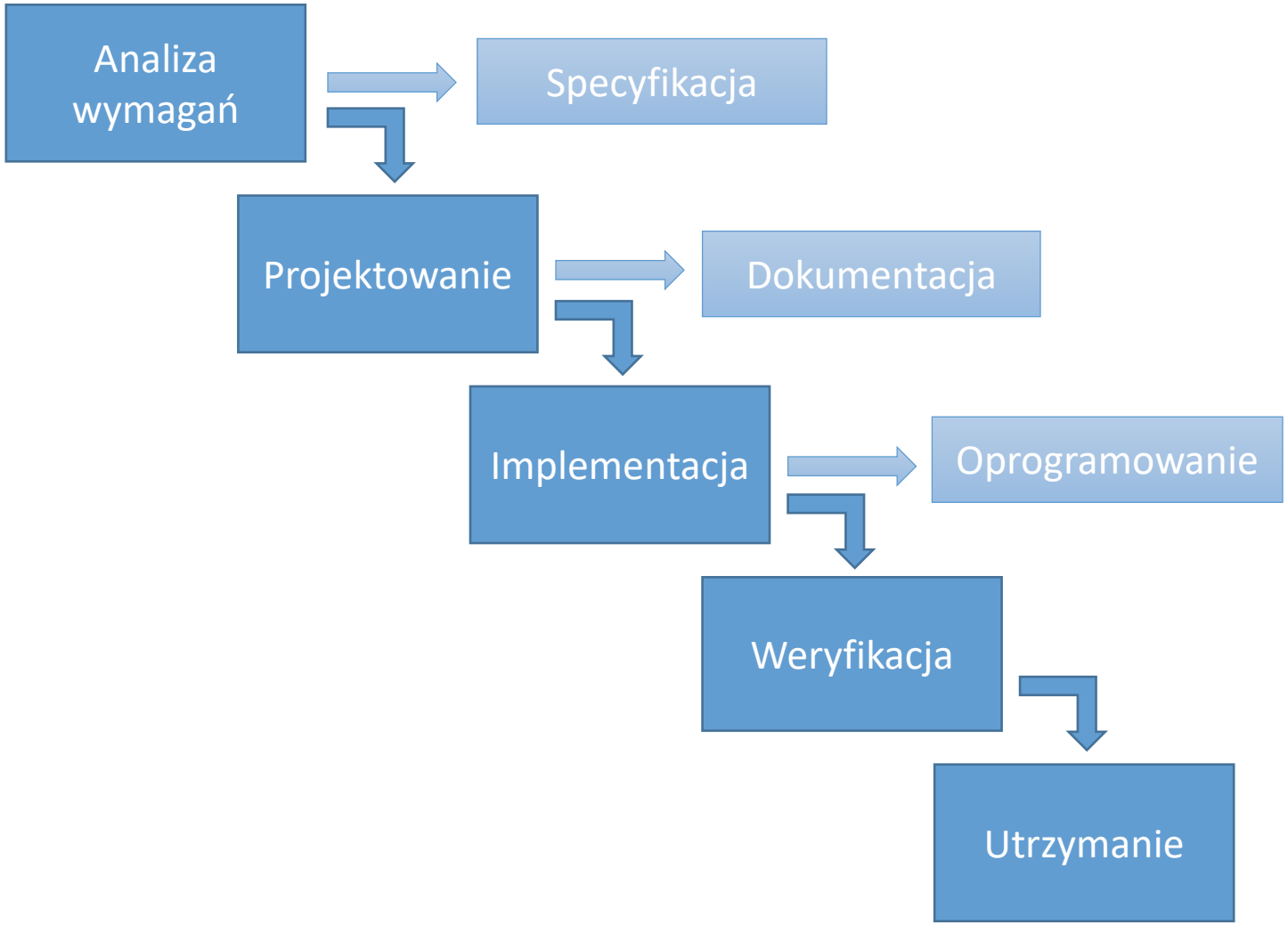
- Etap rozwoju oprogramowania niekiedy traktowany pobieżnie
  - „są wymagania jakieś to robimy, zobaczymy co wyjdzie”
    (to ani agile, ani extreeme programming)
- Faza kluczowa dla ustalenia celu projektu i zdefiniowania co usatysfakcjonuje klienta
- Dobre wykonanie analizy często wymaga zestawienia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



# Co dalej?

- Zebranie wymagań to pierwszy krok
  - Nawet jeśli wykonywany wielokrotnie w cyklu
- Istnieją różne *modele* realizacji cyklu życia oprogramowania

# Model wodospadowy (waterfall)





# Waterfall - cechy

- Dzieli proces na łatwe do zrozumienia i rozdzielenia etapy
- Dostarcza pośrednie efekty pracy
- ... i zdaniem wielu - nie działa (otwarta pętla sterowania)
  - czy też – istnieje, by udowadniać, że inne metody są lepsze
- Ale jest domyślnym sposobem postępowania przy kontraktach o „ustalonej cenie” (*fixed price*)
  - szczególnie w szeroko rozumianej tzw. sferze budżetowej
  - duże doświadczenie wytwórcy w konkretnych problemach *może* działać
- Dlatego bardzo często występuje w formach *zmodyfikowanych*
  - Jakieś formy iteracji, podzielenia na pod-problemy, nachodzenia faz etc.

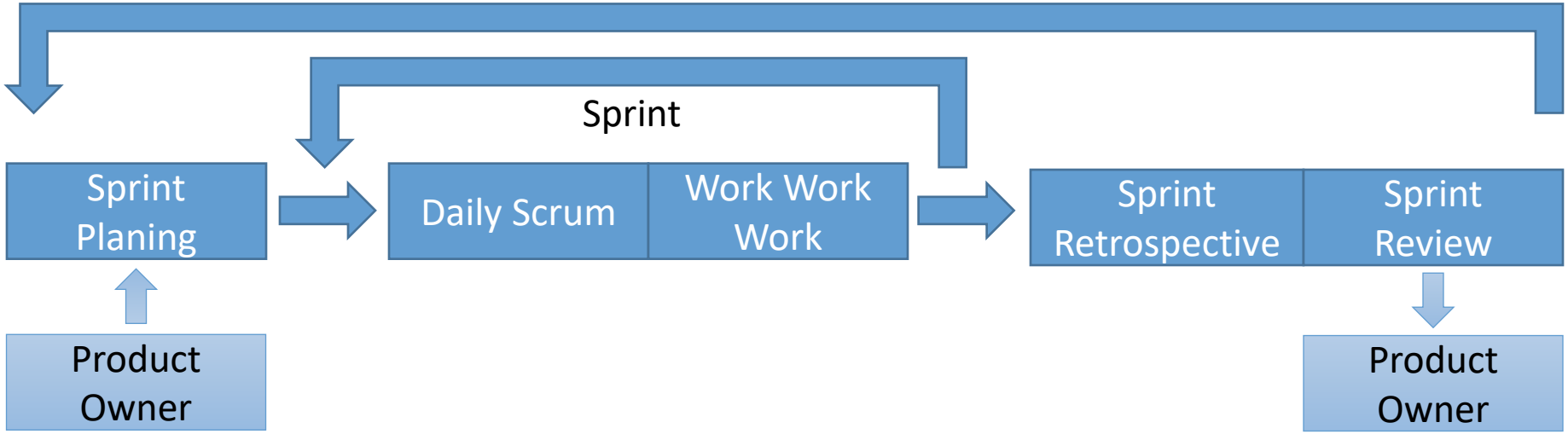


# Metodyki zwinne (Agile)

- W pewnym sensie – drugie ekstremum
- Krótkie cykle „od wymagań do wdrożenia”
  - liczone w dniach, nie miesiącach
- Interakcja z klientem na każdym etapie
- Oprogramowanie praktycznie non-stop „gotowe do wdrożenia”
  - To powiązane z CI/CD (ale CI to narzędzie, a nie metodyka)
- Hipotetycznie – *zwinna* reakcja na zmiany / elastyczność
- ... ale wymaga tej elastyczności także od klienta
- ... także w kosztach

# SCRUM

- Krótkie cykle – Sprint (zazwyczaj 2 – 4 tyg.)
- Klient / reprezentant interesów klienta – Product Owner
- Opiekun i „kapitan” zespołu – Scrum Master
- *Codzienna* obserwacja procesu





# Tak dla odmiany ...

- ... nie ma procesu z definicji *najlepszego*
- Agile wydaje się „naturalniejszy”
  - Sprawniej radzi sobie z krótkoterminowymi problemami
- Ale jak np. *zacząć* projekt sprintami?
  - To problem wielu przyrostowych modeli
- Co z projektami będącymi częścią większego przedsięwzięcia?
  - Terminy, konieczna wymiana dokumentacji etc.
- Co z projektami o ustalonym budżecie?
  - Zarówno Waterfall jak i Agile twierdzą, że obniżają koszty tego drugiego
- Często występuje w efekcie jakaś mieszanka
  - Osobiście *marzy mi się* prawdziwy SCRUM ze *świadomym* klientem
- Dostosowywać proces do potrzeb, a nie potrzeby do procesu



# Agile Manifesto

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

**Individuals and interactions** over processes and tools
**Working software** over comprehensive documentation
**Customer collaboration** over contract negotiation
**Responding to change** over following a plan

That is, while *there is value* in the items on
the right, we value the items on the left more.



# Cykl życia c.d.

- Mamy wymagania
- Mamy *hipotezy/modele* jak pracować by przetworzyć je w kolejne wersje oprogramowania
- Warto teraz porozmawiać o elemencie 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 – dwa użycia

- Oddzielny element cyklu, dedykowana czynność
  - Czasem zwany „walidacją”, „kwalifikacją”, „certyfikacją” etc.
- Integralny element „implementacji” / codziennej pracy
  - Nie można ocenić wykonanej pracy bez jej przetestowania
- Oba mają sens

# Dziękuję za uwagę

Konrad.Grochowski@pw.edu.pl

