

Wykład nr 1

Temat: Cykle życia i procesy wytwarzania oprogramowania

Paweł Karczmarek

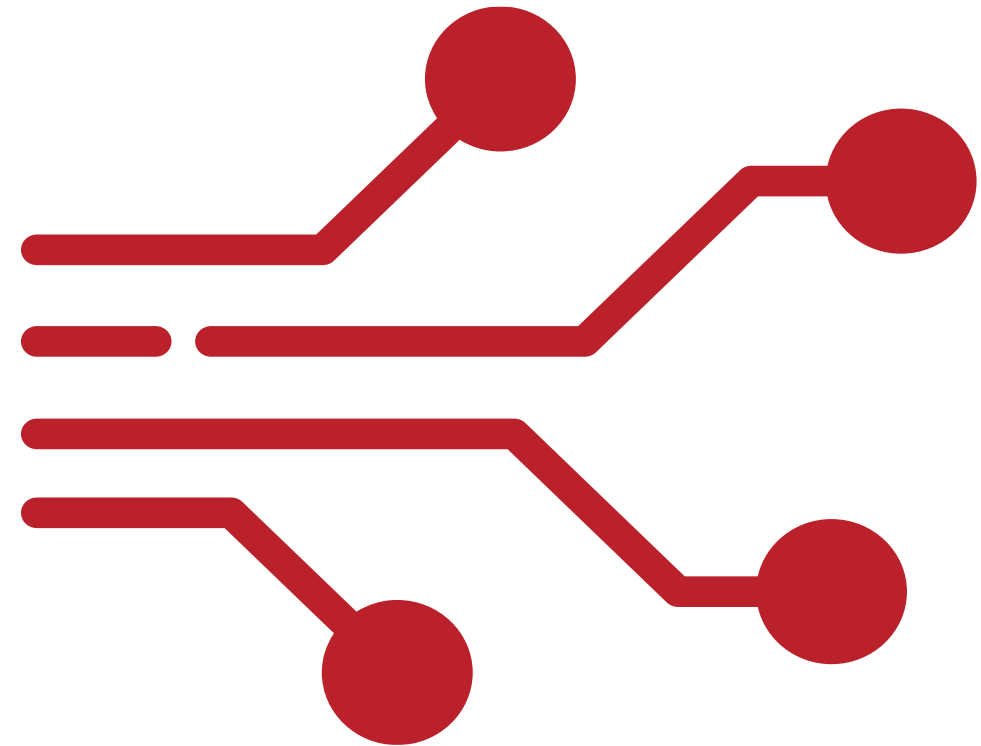
Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

Plan prezentacji

- Czym jest inżynieria oprogramowania?
- Cykl życia oprogramowania
- Proces



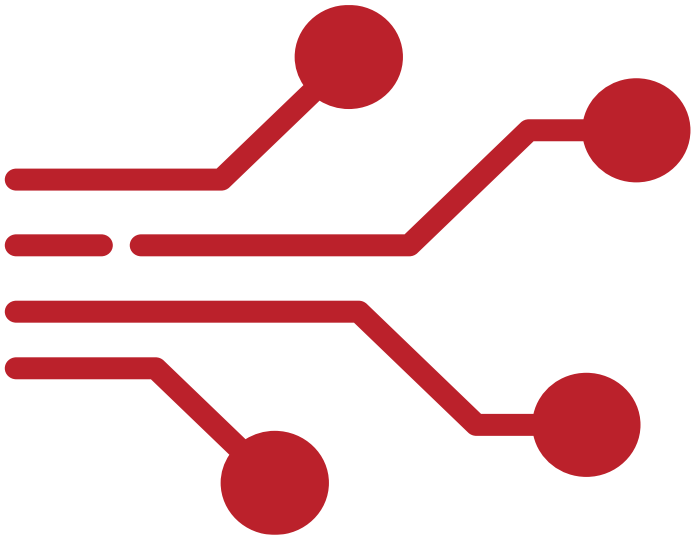
Czym jest inżynieria oprogramowania?



3

Oprogramowanie jako „niewidzialna” konstrukcja inżynierska

- Oprogramowanie nie ma fizycznej postaci
- Może osiągać ogromną skalę złożoności
- Struktura systemu jest niewidoczna dla użytkownika
- Zależności między elementami są logiczne, a nie materialne
- Błąd w jednym miejscu może mieć konsekwencje globalne



Dlaczego tworzenie oprogramowania jest trudniejsze niż się wydaje?

- Złożoność rośnie szybciej niż liczba funkcji
- Wymagania zmieniają się w czasie
- System działa w dynamicznym otoczeniu
- Tworzenie odbywa się zespołowo
- Skutki błędów mogą być nieproporcjonalnie duże



Wzrost złożoności współczesnych systemów informatycznych

- Systemy rozproszone i wielowarstwowe
- Integracja z usługami zewnętrznymi
- Przetwarzanie w czasie rzeczywistym
- Skalowalność do tysięcy lub milionów użytkowników
- Współistnienie różnych technologii i platform



Kryzys oprogramowania

- Dynamiczny rozwój sprzętu w latach sześćdziesiątych
- Rosnąca skala i złożoność systemów
- Projekty znacząco przekraczające budżety i harmonogramy
- Niska niezawodność i trudności w utrzymaniu
- Konferencja NATO w 1968 roku (Garmisch) jako punkt zwrotny
- Narodziny inżynierii oprogramowania jako dyscypliny

7

Geneza inżynierii oprogramowania

- Odejście od improwizacji na rzecz metody
- Uznanie programowania za działalność inżynierską
- Wprowadzenie modeli cyklu życia
- Rozwój metod analizy i projektowania
- Standaryzacja dokumentacji i kontroli jakości



Czy kryzys oprogramowania został rozwiązany

- Projekty IT nadal przekraczają budżety i harmonogramy
- Wysoka złożoność współczesnych systemów
- Uzależnienie państw i gospodarki od oprogramowania
- Rosnąca liczba incydentów i podatności bezpieczeństwa
- Stała presja na skracanie czasu wytwarzania



Definicja inżynierii oprogramowania

- Systematyczne podejście do tworzenia oprogramowania
- Obejmuje cały cykl życia systemu
- Łączy aspekty techniczne, organizacyjne i jakościowe
- Uwzględnia ograniczenia czasu, budżetu i zasobów
- Zakłada świadome zarządzanie iem

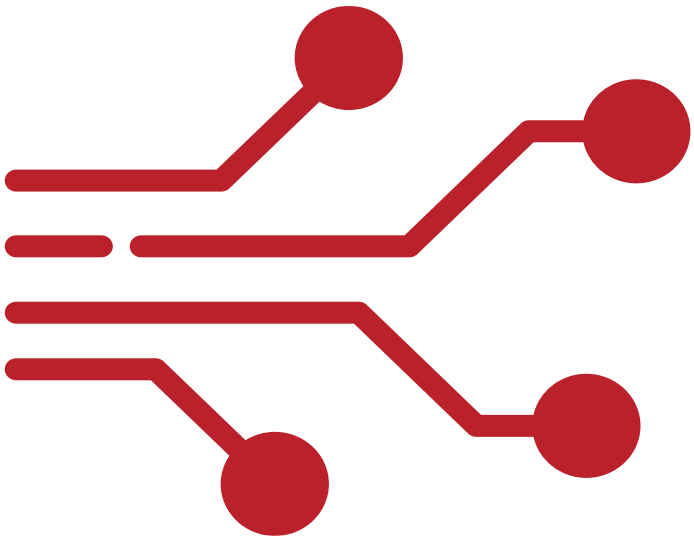


Informatyka a inżynieria oprogramowania

- Informatyka bada podstawy obliczeń i algorytmów
- Inżynieria oprogramowania koncentruje się na budowie systemów
- Teoria a praktyka organizacyjna
- Model matematyczny a produkt działający w środowisku rzeczywistym
- Rozwiązanie optymalne a rozwiązanie wykonalne

Inżynieria oprogramowania a inżynieria systemów

- System jako całość: oprogramowanie, sprzęt, ludzie i procedury
- Integracja wielu komponentów w jedno działające rozwiązanie
- Współzależność elementów technicznych i organizacyjnych
- Odpowiedzialność za funkcjonowanie całego systemu
- Szczególne znaczenie w systemach wbudowanych i krytycznych

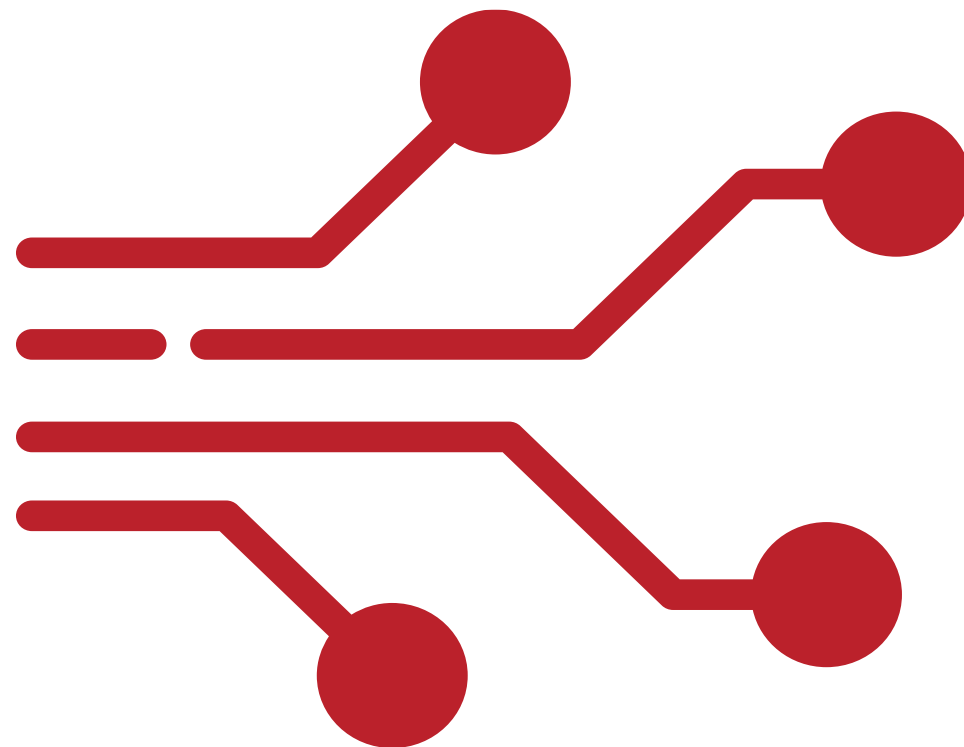


Rodzaje produktów programowych

- Oprogramowanie tworzone na zamówienie
- Produkty masowe dostępne na rynku
- Systemy wbudowane
- Systemy krytyczne
- Usługi oparte na oprogramowaniu



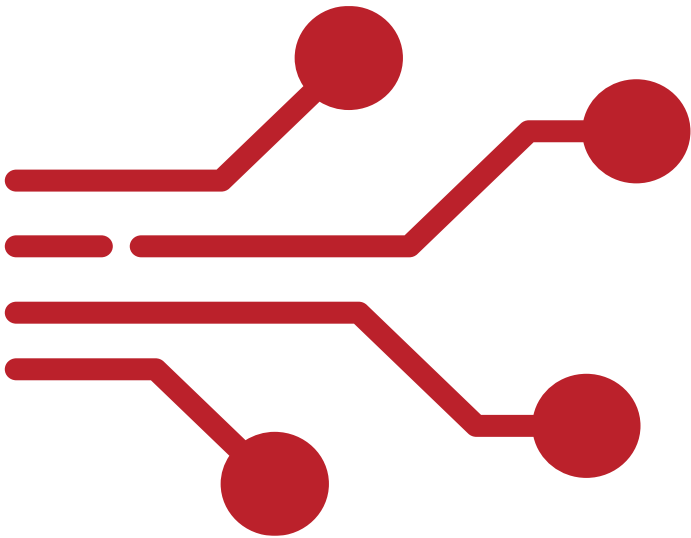
Cykl życia oprogramowania



14

Cykl życia oprogramowania

Cykl życia oprogramowania to uporządkowany zbiór etapów, przez które przechodzi oprogramowanie od momentu powstania koncepcji jego budowy, poprzez analizę, projektowanie, implementację i wdrożenie, aż do eksploatacji, modyfikacji oraz ostatecznego wycofania z użycia.



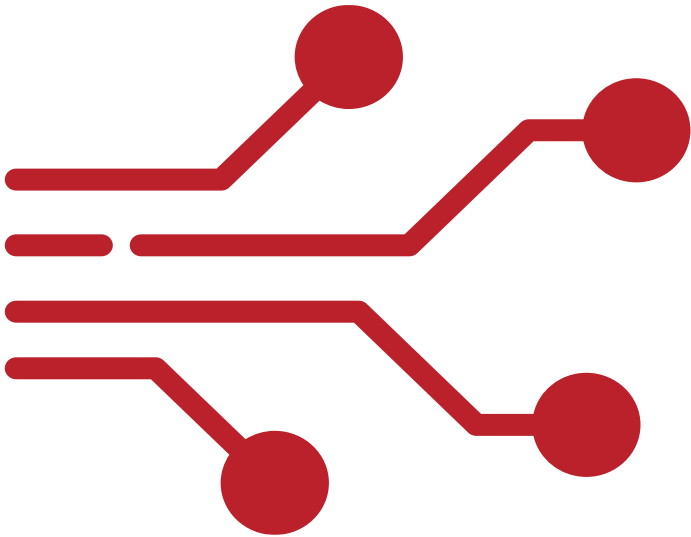
Główne działania w cyklu życia oprogramowania

- Analiza
- Projektowanie
- Implementacja
- Weryfikacja i zatwierdzanie
- Wdrożenie
- Konserwacja



Koszty w cyklu życia oprogramowania

- Koszt błędu rośnie wraz z etapem jego wykrycia
- Najtańsze są błędy usunięte na etapie analizy
- Najdroższe są błędy wykryte po wdrożeniu
- Utrzymanie i rozwój często przewyższają koszt wytworzenia
- Decyzje wczesne determinują koszty późniejsze



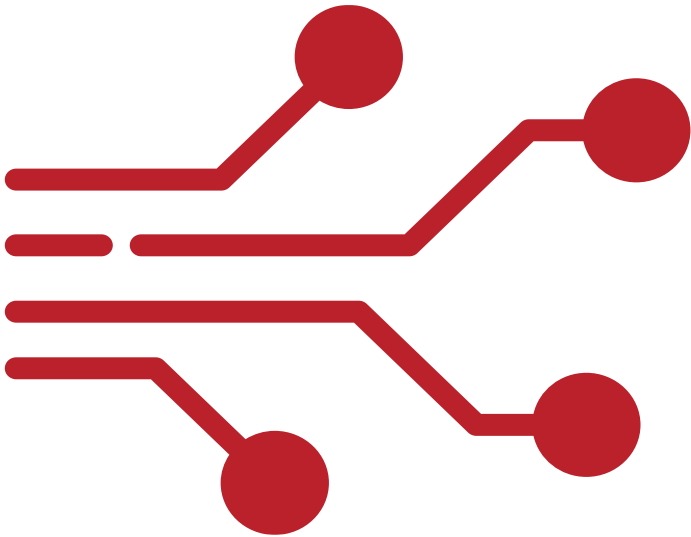
Weryfikacja i zatwierdzanie

- Weryfikacja odpowiada na pytanie: czy zbudowaliśmy system poprawnie
- Zatwierdzanie odpowiada na pytanie: czy zbudowaliśmy właściwy system
- Weryfikacja dotyczy zgodności z projektem i specyfikacją
- Zatwierdzanie dotyczy zgodności z potrzebami użytkownika
- Oba działania są niezbędne dla jakości



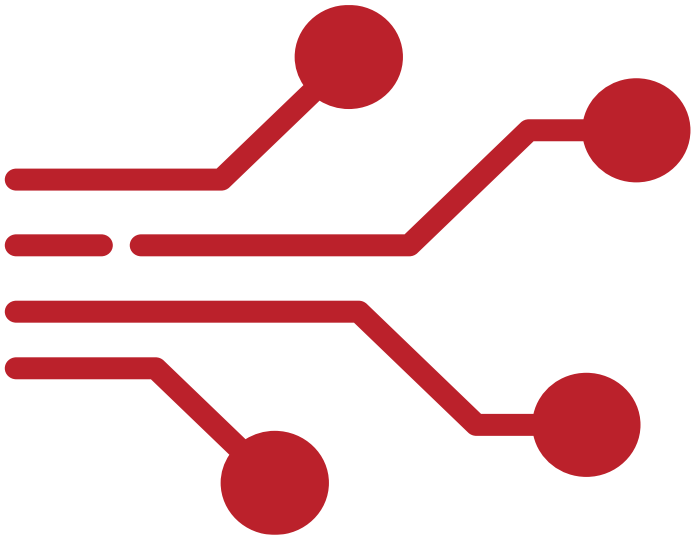
Konserwacja oprogramowania jako dominująca faza cyklu życia

- Usuwanie błędów wykrytych po wdrożeniu
- Dostosowywanie do zmian prawnych i organizacyjnych
- Rozwój nowych funkcjonalności
- Aktualizacje technologiczne
- Aktualizacje związane z bezpieczeństwem



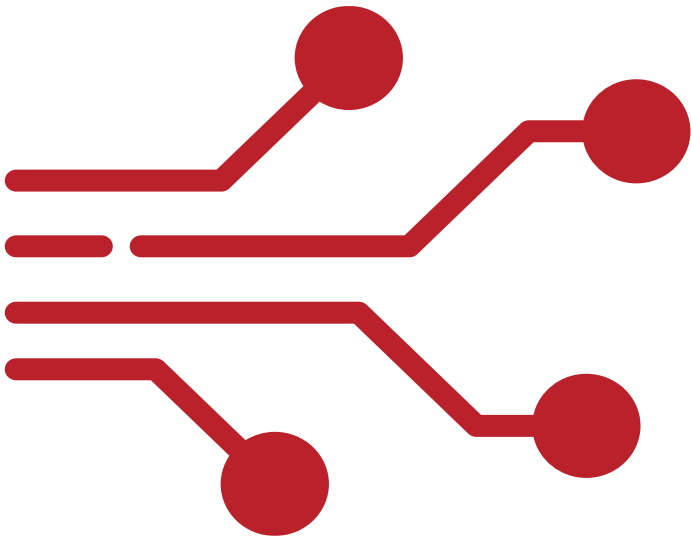
Model procesu wytwarzania oprogramowania

- Uproszczona reprezentacja sposobu organizacji prac
- Określa kolejność działań i ich zależności
- Definiuje role i odpowiedzialności
- Wskazuje produkty pośrednie i końcowe
- Ułatwia planowanie i kontrolę projektu



Model naturalny (buduj i poprawiaj) wytwarzania oprogramowania - struktura

- Pomysł lub potrzeba jako punkt startowy
- Bezpośrednie przejście do implementacji
- Brak formalnej specyfikacji wymagań
- Brak wyraźnego etapu projektowania
- Testowanie wykonywane po napisaniu kodu
- Modyfikacje wprowadzane na bieżąco



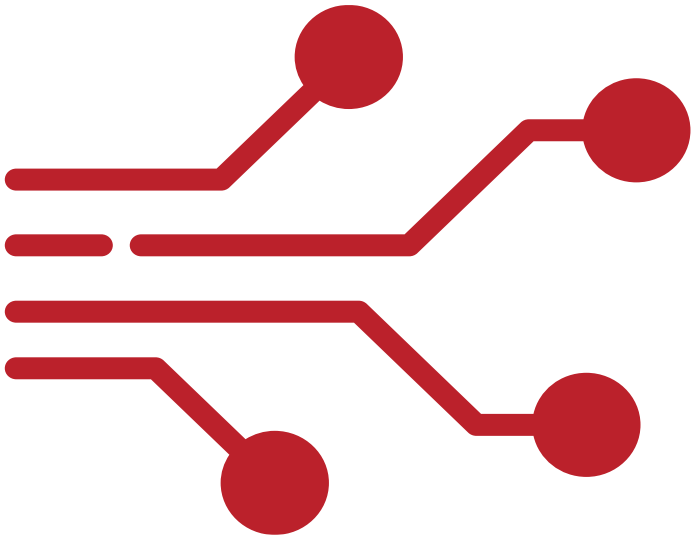
Model naturalny - zalety i ograniczenia

- Zalety
 - Szybkie rozpoczęcie prac
 - Niska bariera formalna
 - Duża elastyczność na wczesnym etapie
 - Przydatny w małych projektach i prototypowaniu
- Ograniczenia
 - Brak kontroli nad zakresem i strukturą systemu
 - Narastanie chaosu architektonicznego
 - Trudności w pracy zespołowej
 - Wysokie o błędów i podatności
 - Problemy z utrzymaniem i rozwojem systemu

22

Model kaskadowy - struktura

- Analiza i specyfikacja wymagań
- Projektowanie systemu
- Implementacja
- Integracja i testowanie
- Wdrożenie
- Utrzymanie



Model kaskadowy - cechy charakterystyczne

- Ścisła sekwencyjność etapów
- Wyraźne granice między fazami
- Formalne zatwierdzanie wyników każdej fazy
- Silny nacisk na dokumentację
- Planowanie całości projektu na początku
- Ograniczona adaptacyjność do zmian



Model kaskadowy (wodospadowy) - zalety i ograniczenia

- **Zalety**

- Przejrzysta i uporządkowana struktura procesu
- Łatwość planowania harmonogramu i budżetu
- Jasne punkty kontrolne i odpowiedzialność
- Rozbudowana dokumentacja ułatwiająca audyt
- Dobrze sprawdza się przy stabilnych wymaganiach

- **Ograniczenia**

- Trudność w reagowaniu na zmiany wymagań
- Wysoki koszt powrotu do wcześniejszych faz
- Późne uzyskanie informacji zwrotnej od użytkownika
- o budowy systemu nie w pełni odpowiadającego realnym potrzebom
- Mała elastyczność w dynamicznym środowisku

25

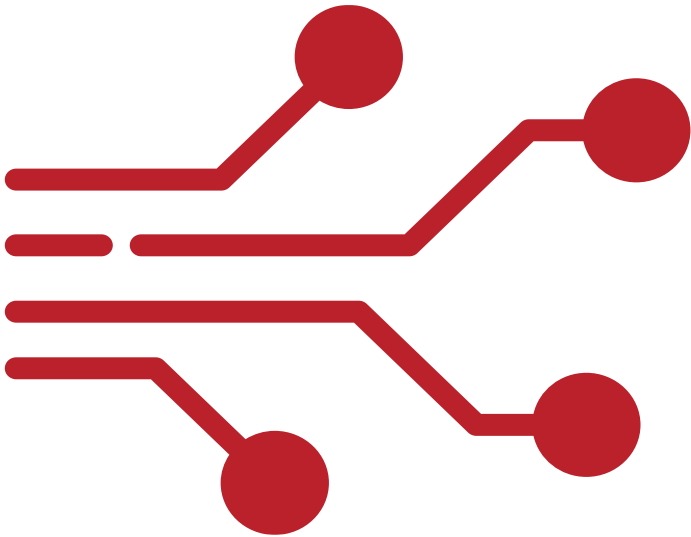
Model iteracyjny

- Proces realizowany w powtarzających się cyklach
- Każda iteracja obejmuje analizę, projektowanie, implementację i testowanie
- Stopniowe doskonalenie systemu
- Wczesne uzyskiwanie informacji zwrotnej
- Możliwość korekty kierunku rozwoju



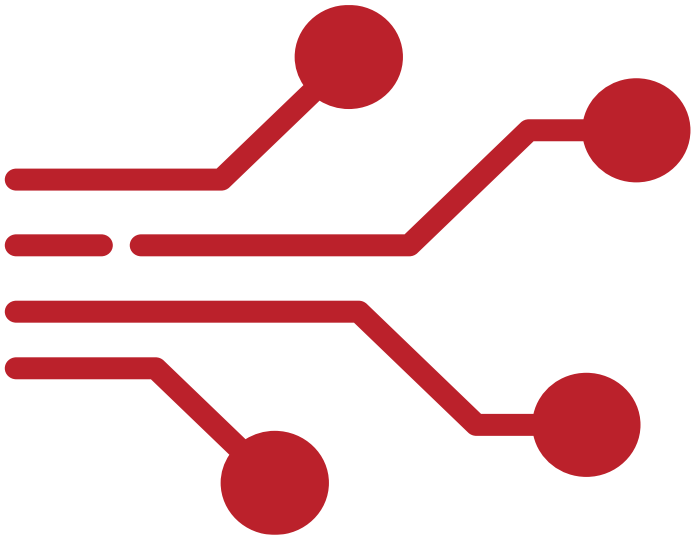
Model przyrostowy

- System budowany w kolejnych przyrostach funkcjonalnych
- Każdy przyrost stanowi działającą część systemu
- Rozszerzanie systemu o kolejne moduły
- Wczesne dostarczanie wartości użytkownikowi
- Architektura projektowana z myślą o rozbudowie



Model spiralny

- Proces realizowany w kolejnych cyklach zwanych spiralami
- Każdy cykl obejmuje analizę celów, ocenę a i wytworzenie rozwiązania
- Analiza a jako element centralny
- Możliwość wczesnego prototypowania
- Decyzja o kontynuacji projektu po każdej spirali



Prototypowanie

- Budowa uproszczonej wersji systemu
- Weryfikacja założeń i wymagań
- Lepsze zrozumienie potrzeb użytkownika
- Eksperymentowanie z rozwiązaniami technicznymi
- Możliwość odrzucenia prototypu po spełnieniu celu



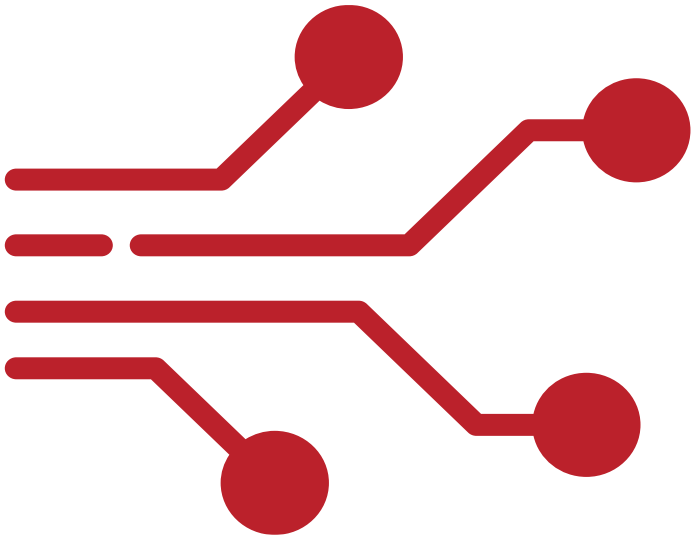
Model V

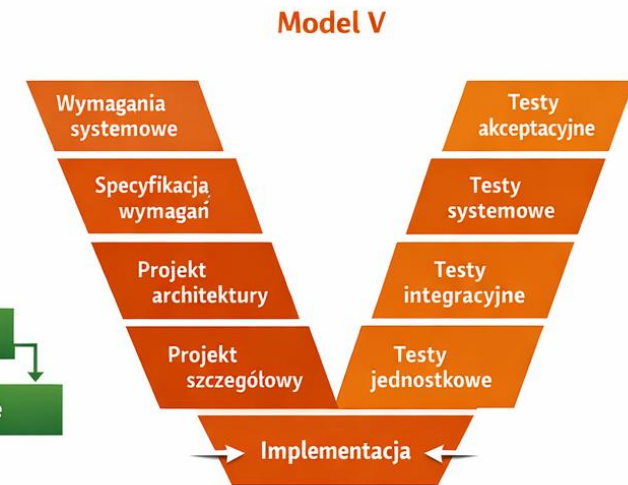
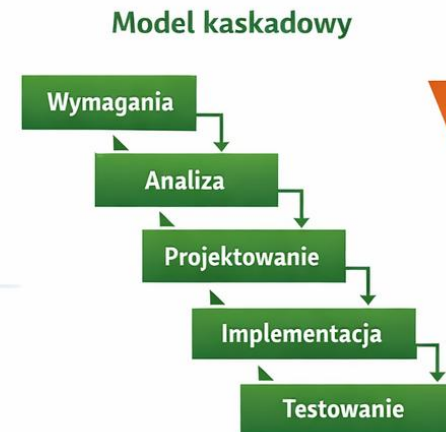
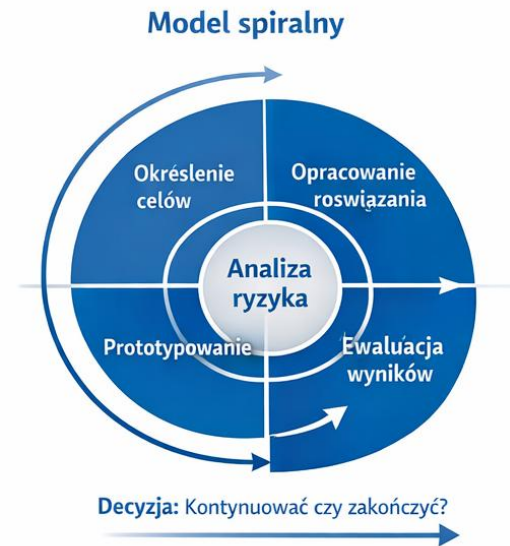
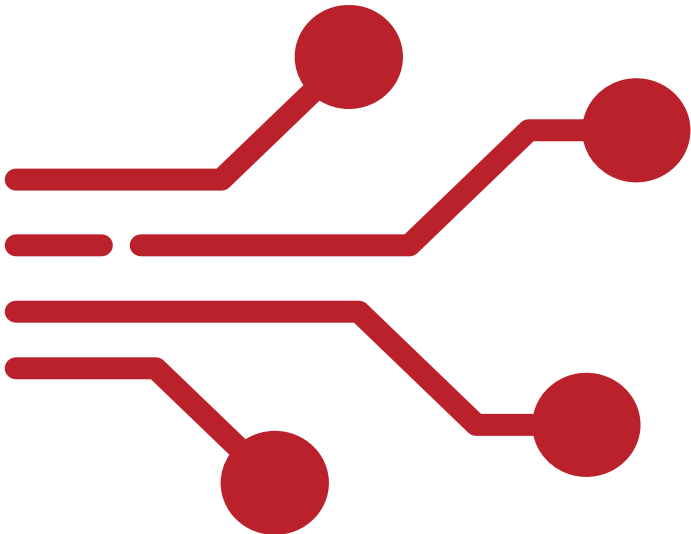
- Rozwinięcie modelu kaskadowego
- Powiązanie etapów projektowych z etapami testowania
- Wczesne planowanie weryfikacji i walidacji
- Każdemu poziomowi specyfikacji odpowiada poziom testów
- Silny nacisk na kontrolę jakości



Model V - korespondujące etapy

- Wymagania systemowe - Testy akceptacyjne
- Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych - Testy systemowe
- Projekt architektury systemu - Testy integracyjne
- Projekt szczegółowy (modułów) - Testy jednostkowe
- Implementacja - Wykonanie testów jednostkowych





Model zwinny (Agile)

- Krótkie iteracje o stałej długości
- Priorytetyzowany zbiór wymagań
- Częste dostarczanie działającego oprogramowania
- Stała współpraca z użytkownikiem
- Adaptacja do zmian zamiast sztywnego planu



Gdzie stosować poszczególne modele procesu

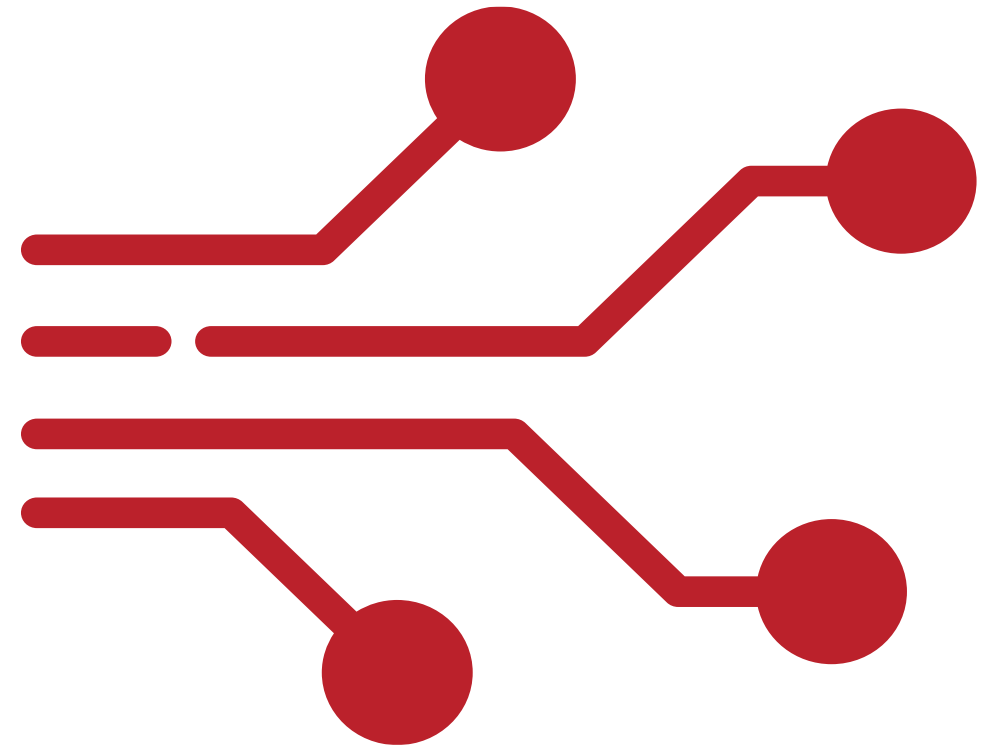
- Model naturalny (małe projekty jednoosobowe, prototypy i eksperymenty, projekty edukacyjne, krótkoterminowe narzędzia pomocnicze)
- Model kaskadowy (stabilne, dobrze zdefiniowane wymagania, projekty kontraktowe o ustalonym zakresie, środowiska silnie regulowane, projekty z formalnym odbiorem etapów)
- Model V (systemy krytyczne, projekty wymagające ścisłej walidacji, oprogramowanie medyczne, przemysłowe, lotnicze, projekty podlegające audytom i normom jakości)

Gdzie stosować poszczególne modele procesu

- Model iteracyjny/przyrostowy (projekty o zmieniających się wymaganiach, systemy rozwijane etapami, duże systemy budowane modułowo, projekty wymagające wczesnej informacji zwrotnej)
- Model spiralny (projekty wysokiego a, nowe technologie, złożone systemy o dużej niepewności, systemy, w których analiza a jest kluczowa)
- Model zwinny (dynamiczne środowisko biznesowe, częste zmiany wymagań, produkty cyfrowe rozwijane ciągle, zespoły pracujące w krótkich iteracjach)

35

Proces



36

Proces jako narzędzie kontroli a i złożoności

- Proces porządkuje działania zespołu
- Ogranicza niepewność projektową
- Wymusza jawność decyzji
- Ułatwia kontrolę jakości
- Stanowi mechanizm redukcji a



Proces a jakość

- Standaryzacja działań zespołu
- Powtarzalność rezultatów
- Punkty kontrolne i przeglądy
- Wczesna detekcja błędów
- Możliwość doskonalenia procesu



Proces a bezpieczeństwo

- Bezpieczeństwo uwzględniane od początku projektu
- Wczesne wykrywanie podatności
- Systematyczna analiza a
- Kontrola zmian w systemie
- Ograniczanie powierzchni ataku



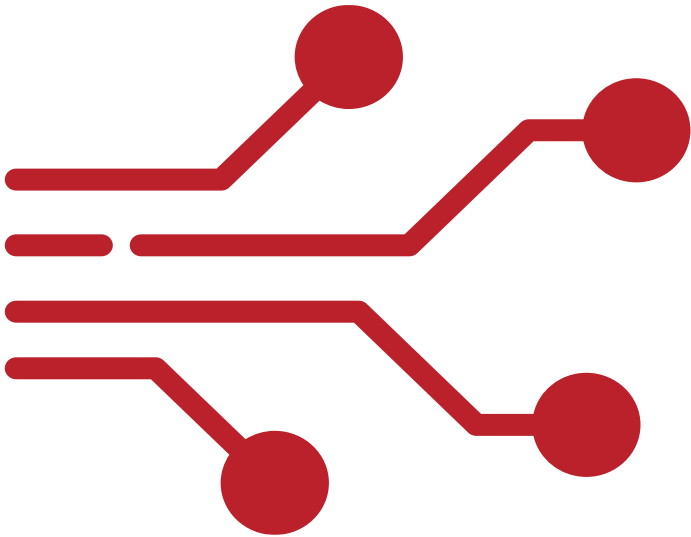
Proces a odpowiedzialność prawna

- Dokumentacja decyzji projektowych
- Rejestrowanie zmian w systemie
- Możliwość odtworzenia historii projektu
- Audytowalność działań
- Zgodność z regulacjami i normami



Ewolucja procesów wytwarzania oprogramowania

- Od modeli ciężkich do adaptacyjnych
- Od dokumentów do działającego oprogramowania
- Integracja testowania i rozwoju
- Automatyzacja budowania i wdrażania
- Ciągłe doskonalenie procesu



Proces jako element kultury organizacyjnej

- Jasno zdefiniowane role i odpowiedzialności
- Standardy komunikacji w zespole
- Świadome zarządzanie iem
- Transparentność działań
- Wspólna odpowiedzialność za jakość i bezpieczeństwo



Podsumowanie

- Proces porządkuje cykl życia oprogramowania
- Wybór modelu zależy od kontekstu projektu
- Proces wspiera jakość, bezpieczeństwo i kontrolę a
- Dokumentacja i kontrola zmian budują odpowiedzialność
- Kolejny krok: inżynieria wymagań



Dziękuję za uwagę



Wykład nr 2 Temat: Inżynieria wymagań. Pozyskiwanie wymagań

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

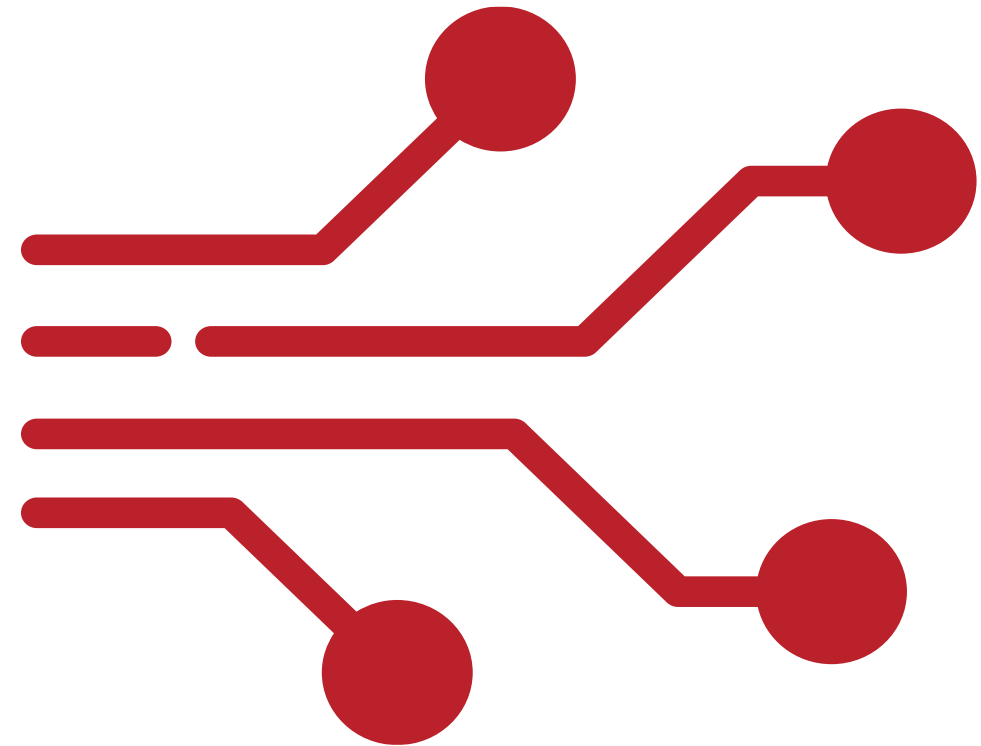
Plan prezentacji

- Wymagania
- Pozyskiwanie wymagań
- Dokumentowanie wymagań



46

Wymagania



Dlaczego wymagania decydują o sukcesie projektu

- Większość błędów ma źródło w fazie wymagań
- Błędne wymagania generują błędną architekturę
- Wymagania determinują testy i kryteria akceptacji
- Koszt błędu rośnie wraz z etapem jego wykrycia
- Bezpieczeństwo zaczyna się od wymagań



Czym jest wymaganie

- Opis oczekiwanej cechy lub zachowania systemu
- Wyraża potrzebę interesariusza
- Stanowi podstawę projektowania i testowania
- Powinno być jednoznaczne i weryfikowalne
- Jest elementem kontraktu projektowego



Potrzeba a wymaganie

- Potrzeba opisuje problem lub cel
- Wymaganie opisuje sposób jego realizacji przez system
- Potrzeba jest często ogólna i biznesowa
- Wymaganie musi być precyzyjne i technicznie interpretowalne
- Mylenie tych pojęć prowadzi do błędów projektowych



Rodzaje wymagań

- Wymagania biznesowe - opisują cel organizacyjny. Odpowiadają na pytanie: dlaczego system powstaje? Mogą dotyczyć zwiększenia efektywności, redukcji kosztów, poprawy bezpieczeństwa czy spełnienia obowiązków prawnych
- Wymagania użytkownika - opisują oczekiwania osób korzystających z systemu. Są zwykle formułowane językiem zrozumiałym dla użytkownika, bez szczegółów technicznych
- Wymagania systemowe - bardziej techniczne. Określają, co system jako całość musi realizować. Stanowią pomost między poziomem biznesowym a projektowym.

51

Rodzaje wymagań

- Wymagania funkcjonalne - opisują konkretne funkcje i zachowania systemu. Co system ma robić w określonych sytuacjach.
- Wymagania niefunkcjonalne - dotyczą jakości działania systemu: wydajności, bezpieczeństwa, niezawodności, użyteczności, skalowalności.
- Wymagania regulacyjne - wynikają z przepisów prawa, norm branżowych lub polityk organizacyjnych. Często mają charakter obowiązkowy i niepodlegający negocjacji.

52

Wymagania biznesowe

- Określają cel projektu
- Wynikają ze strategii organizacji
- Opisują wartość, jaką system ma dostarczyć
- Są mierzone wskaźnikami biznesowymi
- Stanowią punkt odniesienia dla wszystkich dalszych decyzji



Wymagania użytkownika

- Opisują oczekiwania osób korzystających z systemu
- Formułowane językiem zrozumiałym dla użytkownika
- Koncentrują się na zadaniach i celach Nie zawierają szczegółów implementacyjnych
- Stanowią podstawę do dalszej analizy systemowej



Wymagania systemowe

- Precyzują sposób realizacji potrzeb użytkownika
- Formułowane językiem technicznym
- Stanowią podstawę projektowania architektury
- Obejmują funkcje i właściwości jakościowe
- Są bezpośrednio weryfikowalne



Wymagania funkcjonalne

- Określają, co system ma robić
- Opisują reakcje na zdarzenia
- Definiują przetwarzanie danych
- Określają interakcje z użytkownikiem i innymi systemami
- Stanowią podstawę przypadków użycia



Wymagania niefunkcjonalne

- Określają, jak system ma działać
- Dotyczą jakości i właściwości systemu
- Obejmują wydajność, niezawodność, bezpieczeństwo
- Wpływają bezpośrednio na architekturę
- Często są trudniejsze do precyzyjnego sformułowania



Wymagania niefunkcjonalne - podział

- Wymagania produktowe
 - Wydajność
 - Niezawodność
 - Bezpieczeństwo
 - Użyteczność
- Wymagania organizacyjne
 - Standardy i procedury wewnętrzne
 - Narzucone technologie
 - Metodyki i narzędzia
- Wymagania zewnętrzne
 - Regulacje prawne
 - Normy branżowe
 - Wymagania interoperacyjności

Wymagania bezpieczeństwa

- Uwierzytelnianie użytkowników
- Autoryzacja i kontrola dostępu
- Poufność danych
- Integralność informacji
- Rejestrowanie i monitorowanie zdarzeń
- Ochrona przed nadużyciami



Mierzalność i weryfikowalność wymagań

- Wymaganie musi być testowalne
- Unikanie sformułowań ogólnych
- Określanie parametrów liczbowych
- Definiowanie kryteriów akceptacji
- Każde wymaganie powinno mieć sposób weryfikacji



Cechy dobrego wymagania

- Jednoznaczne
- Spójne z innymi wymaganiami
- Kompletne
- Realizowalne
- Weryfikowalne
- Możliwe do śledzenia

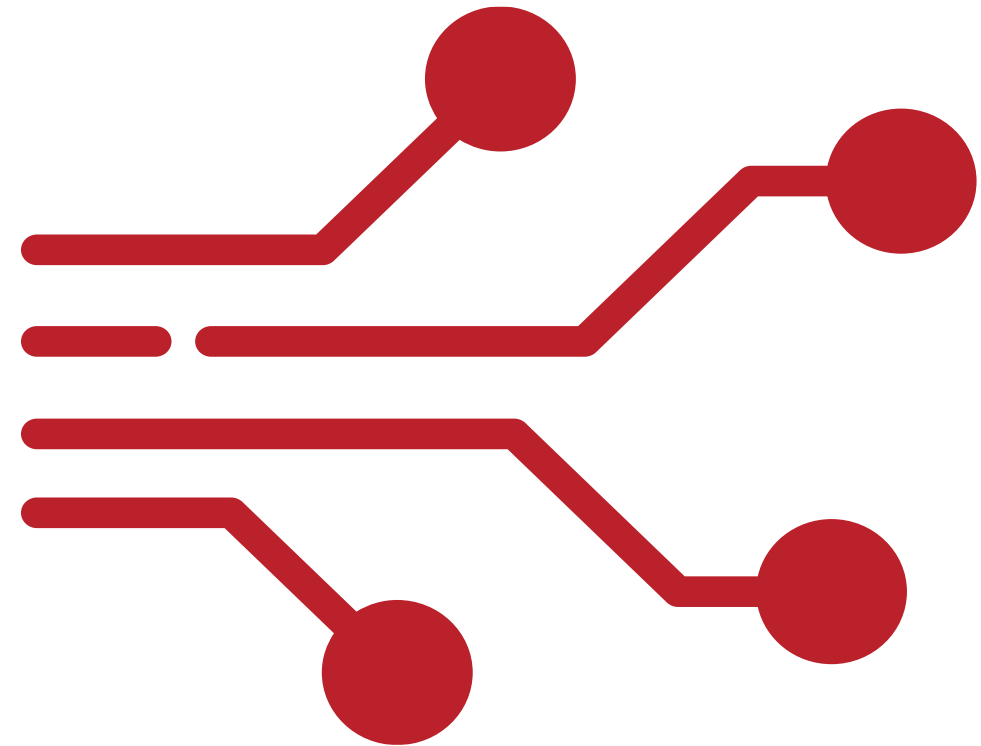


Typowe błędy w wymaganiach

- Niejednoznaczne sformułowania
- Używanie ogólników
- Ukryte założenia
- Sprzeczności między wymaganiami
- Wymagania nierealistyczne
- Mieszanie poziomów abstrakcji

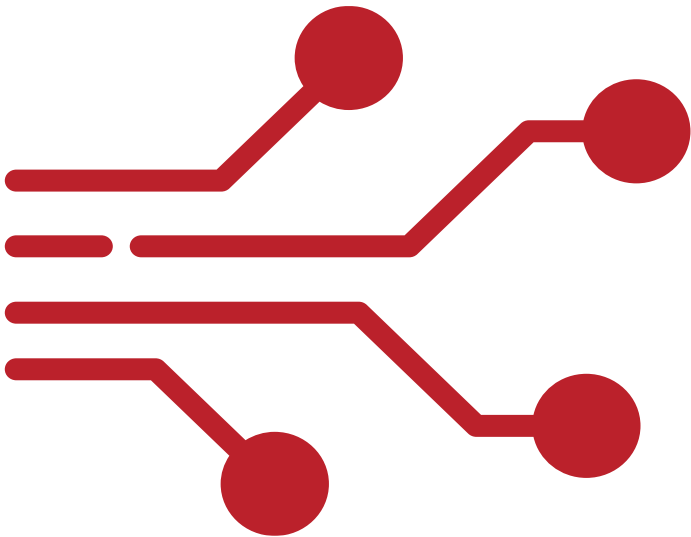


Pozyskiwanie wymagań



Przykład złego wymagania

Zapis:



System powinien być szybki i bezpieczny oraz umożliwiać sprawną obsługę użytkowników.

Przykład poprawionego wymagania

Zapis:

- System musi obsługiwać co najmniej 800 równoczesnych użytkowników.
- Czas odpowiedzi na operację zapisu danych nie może przekraczać 2 sekund w 95% przypadków.
- Dostęp do danych wrażliwych wymaga uwierzytelniania wieloskładnikowego.
- Wszystkie operacje administracyjne muszą być rejestrowane w dzienniku zdarzeń.

65

Proces inżynierii wymagań

- Identyfikacja interesariuszy
- Pozyskiwanie wymagań
- Analiza i doprecyzowanie
- Specyfikacja
- Walidacja
- Zarządzanie zmianą



Pozyskiwanie wymagań - cel i wyzwania

- Zrozumienie rzeczywistych potrzeb interesariuszy
- Identyfikacja wymagań jawnych i ukrytych
- Ujawnienie konfliktów i ograniczeń
- Przekład języka biznesowego na techniczny
- Minimalizacja a nieporozumień



Wywiad jako technika pozyskiwania wymagań

- Bezpośrednia rozmowa z interesariuszem
- Wywiad strukturalny i niestukturalny
- Pytania otwarte i zamknięte
- Dokumentowanie odpowiedzi
- Weryfikacja zrozumienia



Warsztaty i praca grupowa

- Spotkanie wielu interesariuszy jednocześnie
- Wspólne definiowanie potrzeb
- Ujawnianie konfliktów i zależności
- Priorytetyzacja wymagań
- Moderacja jako klucz do sukcesu



Obserwacja użytkowników

- Analiza rzeczywistego sposobu pracy
- Identyfikacja nieformalnych procedur
- Odkrywanie wymagań ukrytych
- Weryfikacja deklarowanych potrzeb
- Lepsze zrozumienie kontekstu systemu



Prototypowanie w pozyskiwaniu wymagań

- Wizualizacja przyszłego systemu
- Ułatwienie komunikacji z użytkownikiem
- Wczesna walidacja założeń
- Odkrywanie brakujących wymagań
- Redukcja a błędnej interpretacji



Przypadki użycia jako forma specyfikacji

- Aktor
- Cel aktora
- Scenariusz główny
- Scenariusze alternatywne
- Warunki początkowe i końcowe



Walidacja wymagań

- Sprawdzenie zgodności z potrzebami interesariuszy
- Identyfikacja braków i sprzeczności
- Przeglądy dokumentacji
- Prototypy i demonstracje
- Wstępne scenariusze testowe



Priorytetyzacja wymagań

- Nie wszystkie wymagania mają tę samą wagę
- Ograniczenia czasu, budżetu i zasobów
- Wymagania krytyczne i opcjonalne
- Ustalanie minimalnego zakresu systemu
- Kompromis między funkcjonalnością a jakością



Priorytetyzacja wymagań

- Nie wszystkie wymagania mają tę samą wagę
- Ograniczenia czasu, budżetu i zasobów
- Wymagania krytyczne i opcjonalne
- Ustalanie minimalnego zakresu systemu
- Kompromis między funkcjonalnością a jakością



Zarządzanie zmianą wymagań

- Wymagania rzadko pozostają niezmiennie
- Formalny proces zgłaszania zmian
- Analiza wpływu zmiany
- Aktualizacja dokumentacji
- Kontrola wersji i decyzji



Śledzalność wymagań (traceability)

- Powiązanie wymagania z jego źródłem
- Powiązanie z elementami projektu
- Powiązanie z testami
- Powiązanie z implementacją
- Możliwość prześledzenia historii zmian



Wymagania a model zagrożeń

- Identyfikacja potencjalnych zagrożeń
- Określenie aktywów wymagających ochrony
- Analiza możliwych scenariuszy ataku
- Definicja wymagań minimalizujących ryzyko
- Powiązanie wymagań z poziomem ryzyka



Wymagania a regulacje i odpowiedzialność

- Wymagania wynikające z przepisów prawa
- Normy i standardy branżowe
- Obowiązki w zakresie ochrony danych
- Odpowiedzialność za naruszenia
- Konieczność dowodowości i audytowalności



Narzędzia wspierające inżynierię wymagań

- Zarządzanie wymaganiami (Jira + dodatki (Requirements, Xray, Zephyr), Azure DevOps, IBM DOORS, Polarion ALM, Jama Connect)
- Modelowanie i wizualizacja (Enterprise Architect, Visual Paradigm, Lucidchart/Draw.io, MagicDraw/Cameo)
- Prototypowanie (Figma, Balsamiq, Adobe XD, Axure RP)
- Dokumentacja i współpraca (Confluence, Notion, Microsoft Teams/Slack)

Przykładowy przepływ pracy z wymaganiami (Jira/Azure DevOps)

- Rejestracja wymagania w backlogu
- Nadanie priorytetu i właściciela
- Analiza i doprecyzowanie
- Powiązanie z zadaniami implementacyjnymi
- Powiązanie z testami
- Zatwierdzenie i zamknięcie



Wymagania w środowisku DevOps i CI/CD

- Ciągła walidacja wymagań
- Automatyczne testy powiązane z wymaganiami
- Integracja kodu z testami bezpieczeństwa
- Monitoring zgodności w czasie rzeczywistym
- Skrócenie czasu reakcji na zmiany

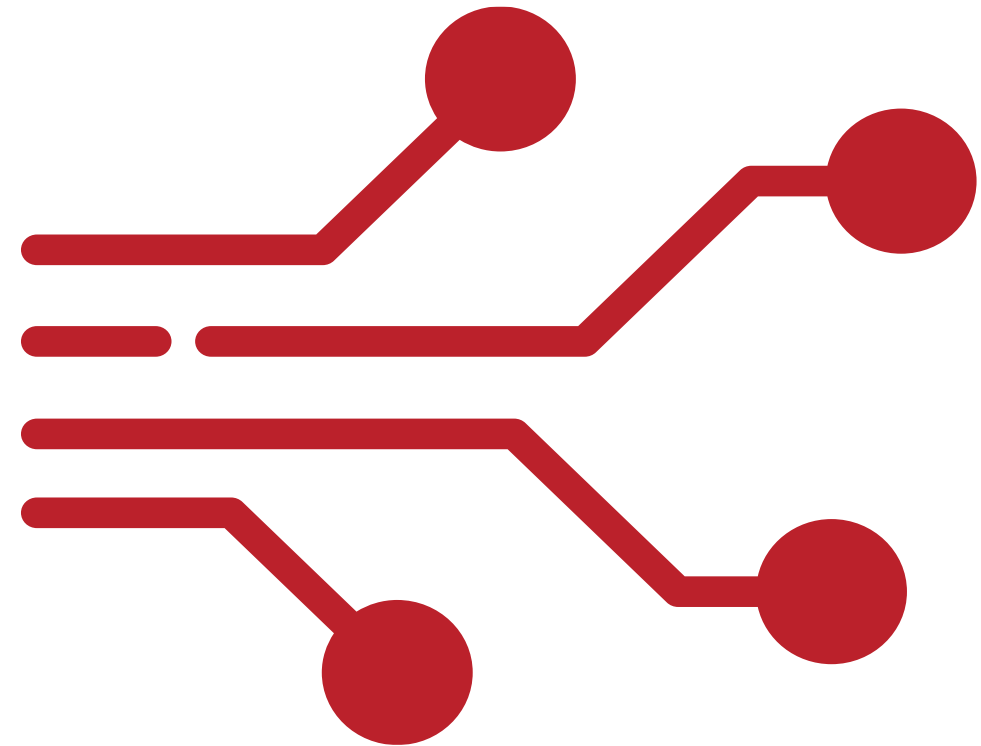


Macierz śledzalności wymagań

- Powiązanie wymagania z przypadkiem użycia
- Powiązanie z modułem systemu
- Powiązanie z testem
- Powiązanie z wersją wdrożenia
- Możliwość raportowania zgodności



Dokumentowanie wymagań



84

Standardy dokumentacji wymagań

- IEEE 29148 – Requirements Engineering
- ISO/IEC/IEEE 12207 – Cykl życia oprogramowania
- ISO/IEC 25010 – Model jakości oprogramowania
- ISO/IEC 27001 – Wymagania bezpieczeństwa informacji
- Specyfikacja SRS (Software Requirements Specification)



ANSI/IEEE 830 - klasyczny standard SRS

- Standard dokumentu Software Requirements Specification
- Opublikowany w 1984, aktualizacja w 1998
- Określał strukturę i zawartość SRS
- Silnie formalny, dokumentocentryczny
- Zastąpiony przez IEEE 29148



IEEE 29148 – Requirements Engineering

- Aktualny standard inżynierii wymagań
- Zastąpił IEEE 830
- Obejmuje wymagania systemowe i programowe
- Definiuje strukturę specyfikacji oraz zasady jakości
- Integruje wymagania z cyklem życia systemu



Struktura dokumentu wg IEEE 29148

- Wprowadzenie: Cel dokumentu. Zakres systemu. Definicje i skróty. Odniesienia
- Ogólny opis systemu: Kontekst systemu. Funkcje na wysokim poziomie. Charakterystyka użytkowników. Ograniczenia i założenia
- Wymagania szczegółowe: Wymagania funkcjonalne. Wymagania нефunkcjonalne. Wymagania interfejsów. Wymagania danych. Wymagania bezpieczeństwa
- Wymagania jakościowe i regulacyjne: Wydajność. niezawodność. Zgodność z normami
- Załączniki: Modele, diagramy. Macierz śledzalności. Historia zmian

88

Podsumowanie – rola inżynierii wymagań

- Wymagania determinują architekturę i testy
- Największy wpływ na koszt i o projektu
- Muszą być jednoznaczne i mierzalne
- Wymagają walidacji i zarządzania zmianą
- Stanowią fundament bezpieczeństwa systemu



Dziękuję za uwagę

Wykład nr 3 Temat: Metody strukturalne analizy i projektowania oprogramowania. Diagramy encji, przepływu danych

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

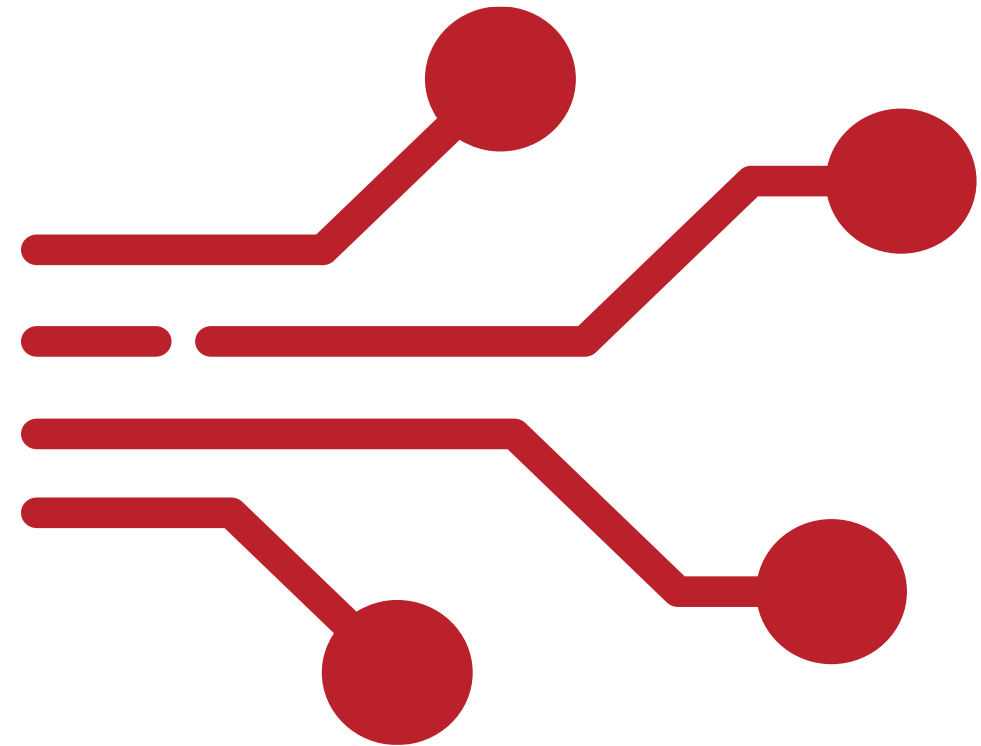
Plan prezentacji

- Wprowadzenie do metod strukturalnych
- Diagram przepływu danych
- Diagram encji
- Integracja modeli



92

Wprowadzenie do metod strukturalnych



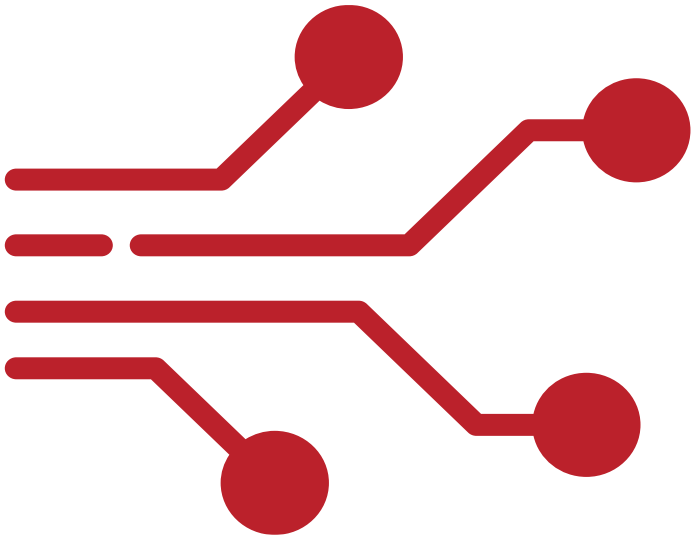
Dlaczego modelujemy system?

- Wymagania są najczęściej zapisane w formie tekstowej
- Tekst bywa niejednoznaczny
- Modele upraszczają złożoność systemu
- Ułatwiają komunikację między interesariuszami
- Stanowią pomost między analizą a implementacją



Metody strukturalne. Podstawowa idea

- System analizowany jest poprzez procesy i dane
- Model opisuje przepływ informacji w systemie
- Funkcje systemu przedstawiane są jako procesy przetwarzania danych
- Dane są modelowane niezależnie od procesów
- Diagramy pomagają uporządkować strukturę systemu



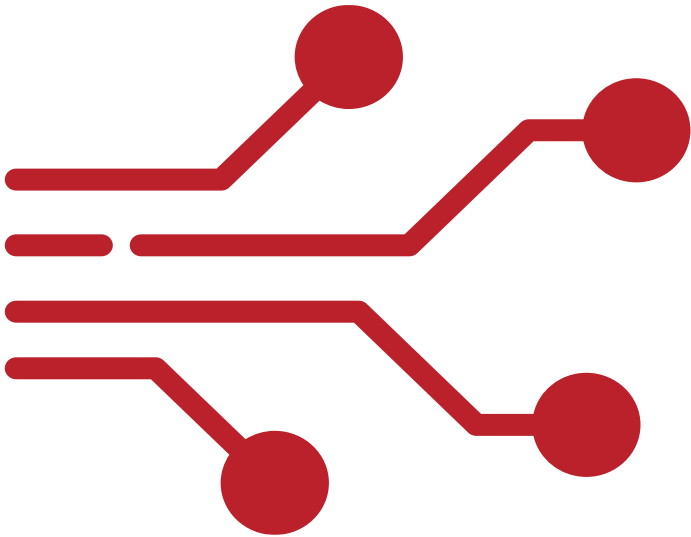
Historia analizy strukturalnej

- Rozwój metod w latach 70. XX wieku
- Odpowiedź na rosnącą złożoność systemów informatycznych
- Powiązanie z programowaniem strukturalnym
- Rozwój metod Yourdona, DeMarco, Gane'a–Sarsona
- Podstawa wielu współczesnych metod modelowania



Analiza strukturalna a projektowanie strukturalne

- Analiza opisuje, co system ma robić
- Projektowanie opisuje, jak system zostanie zbudowany
- Analiza koncentruje się na problemie
- Projektowanie koncentruje się na rozwiązaniu
- Analiza poprzedza etap implementacji



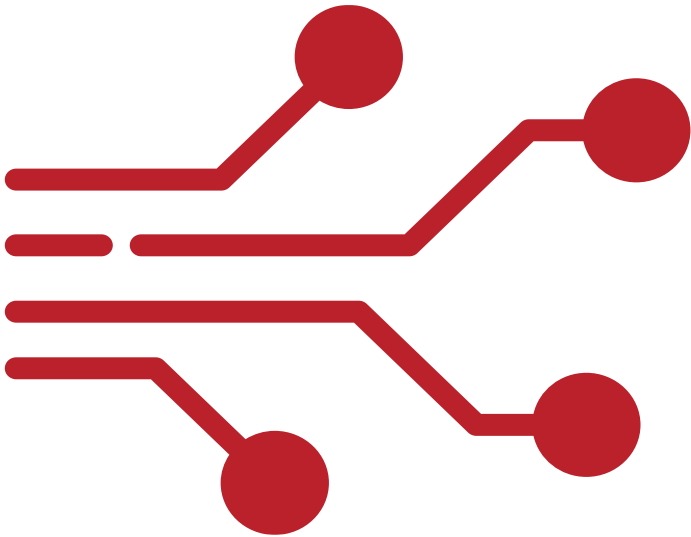
Dane i procesy jako dwa filary systemu

- Każdy system przetwarza dane
- Procesy wykonują operacje na danych
- Dane opisują stan systemu
- Procesy opisują działanie systemu
- Analiza strukturalna modeluje oba elementy



Model funkcjonalny systemu

- Opisuje działania wykonywane przez system
- Przedstawia procesy przetwarzania danych
- Pokazuje przepływ informacji między procesami
- Umożliwia analizę logiki działania systemu
- Najczęściej reprezentowany przez diagramy DFD



Model danych systemu

- Opisuje strukturę informacji w systemie
- Definiuje encje oraz ich atrybuty
- Określa relacje między danymi
- Pozwala zrozumieć organizację informacji
- Najczęściej reprezentowany przez diagramy ER

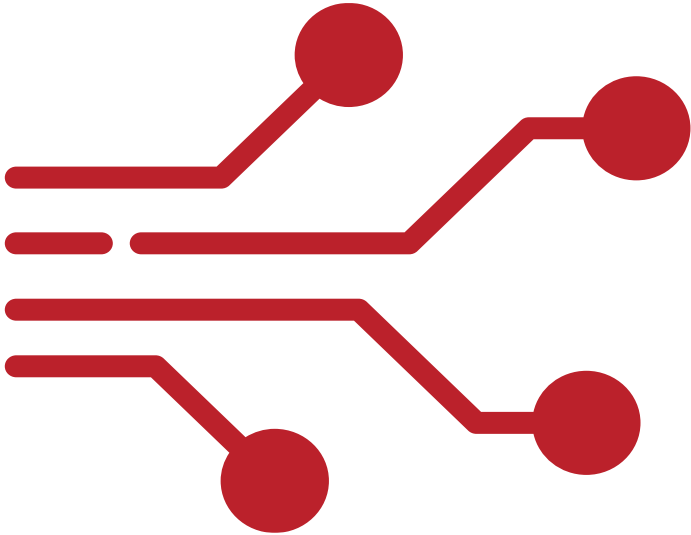
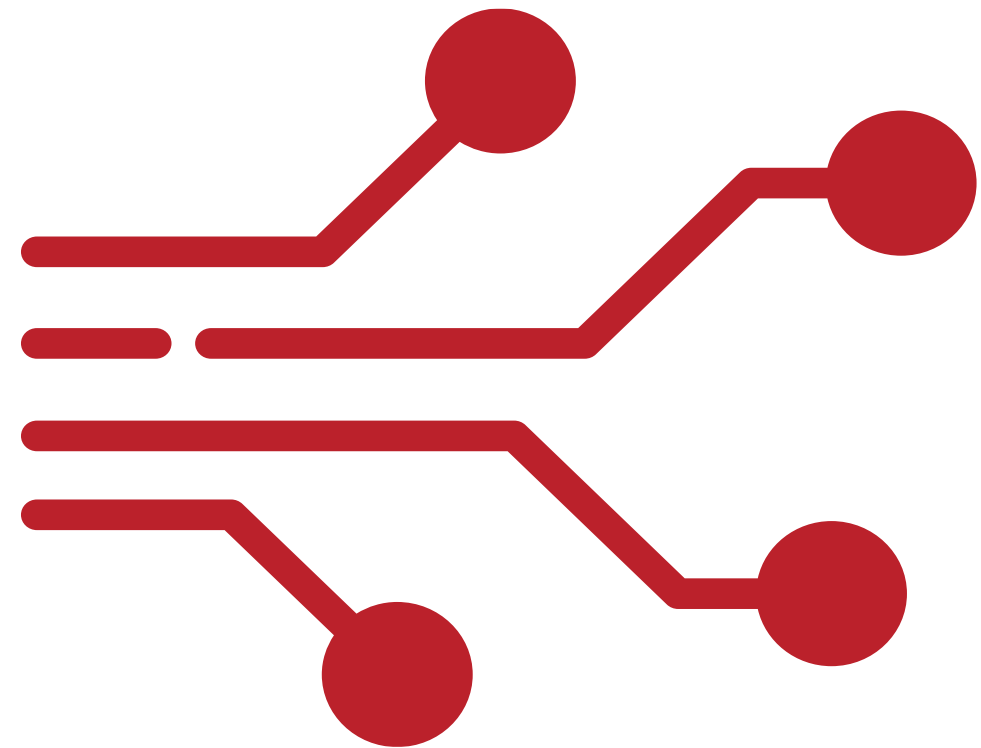


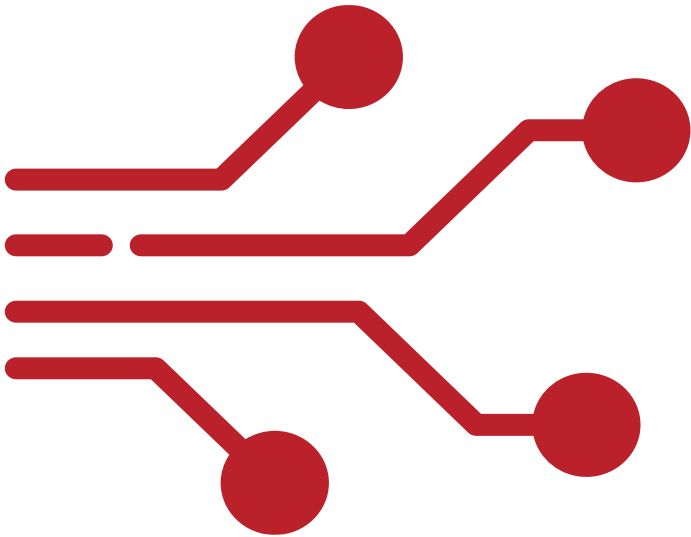
Diagram przepływu danych



101

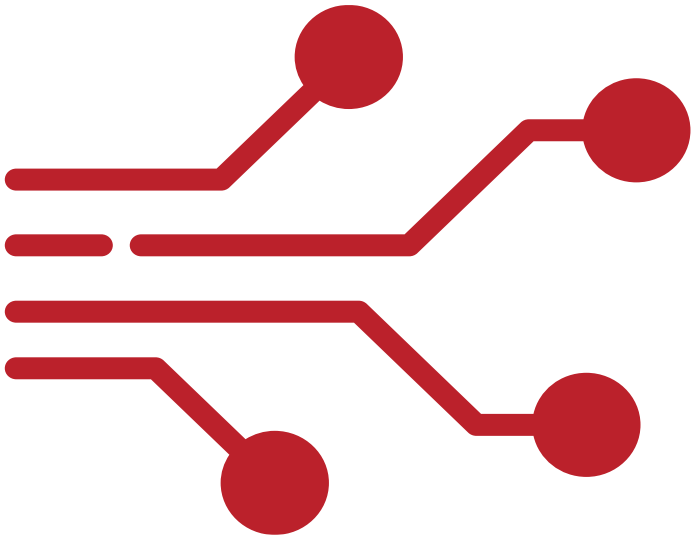
Diagram przepływu danych (DFD) - idea

- Model przedstawiający przepływ informacji w systemie
- Pokazuje procesy przetwarzające dane
- Ilustruje przepływy danych między elementami systemu
- Umożliwia analizę logiki działania systemu
- Stosowany w analizie strukturalnej systemów informatycznych

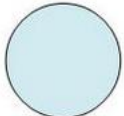


Elementy diagramu DFD

- Proces
- Przepływ danych
- Magazyn danych
- Encja zewnętrzna

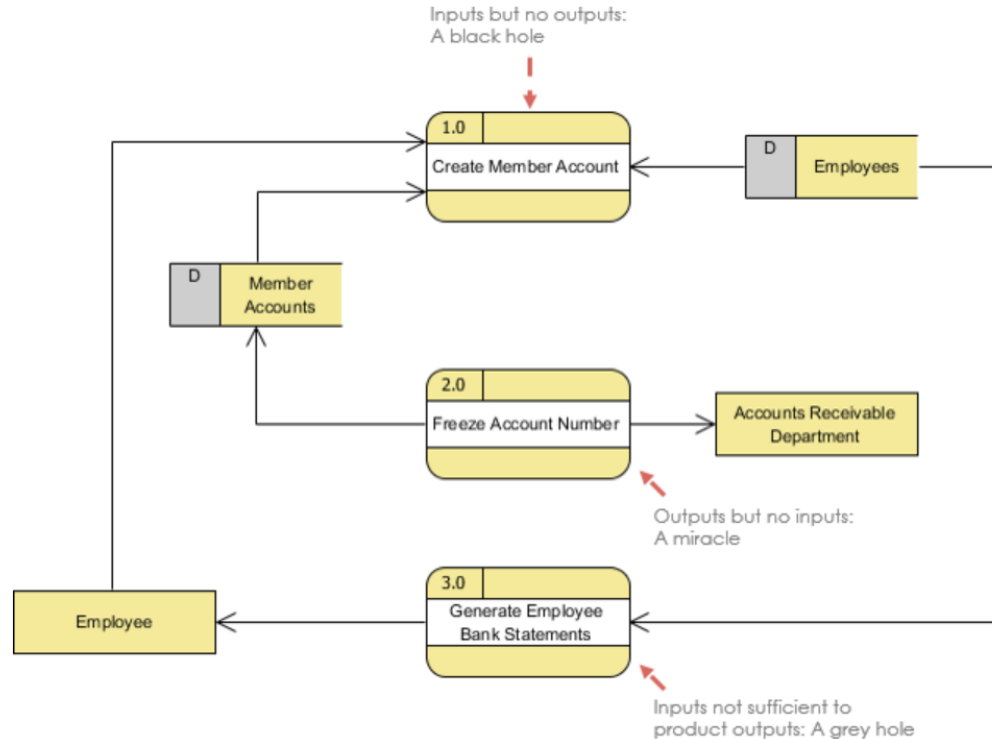


Elementy diagramu DFD

Notation	De Marco & Yourdon	Gane and Sarson
External Entity		
Process		
Data Store		
Data Flow		

Rys. 1. Elementy DFD. Źródło: <https://123projectlab.com/data-flow-diagram-symbols-and-examples/>

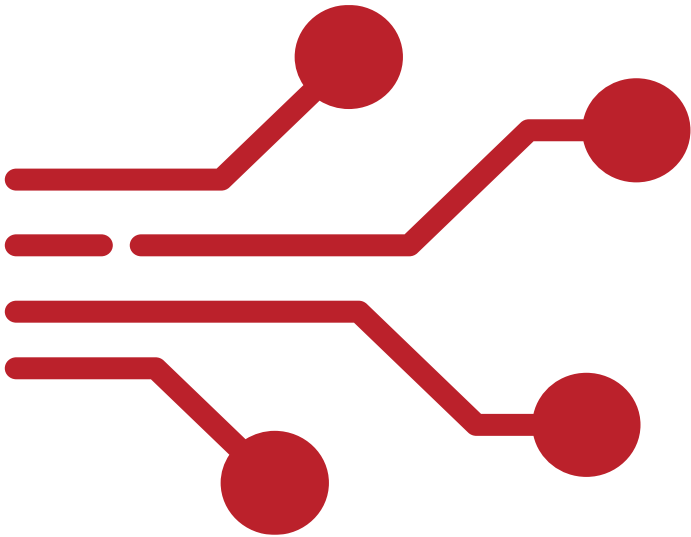
Elementy diagramu DFD



Rys. 2. Źródło: <https://www.visual-paradigm.com/guide/data-flow-diagram/what-is-data-flow-diagram/>

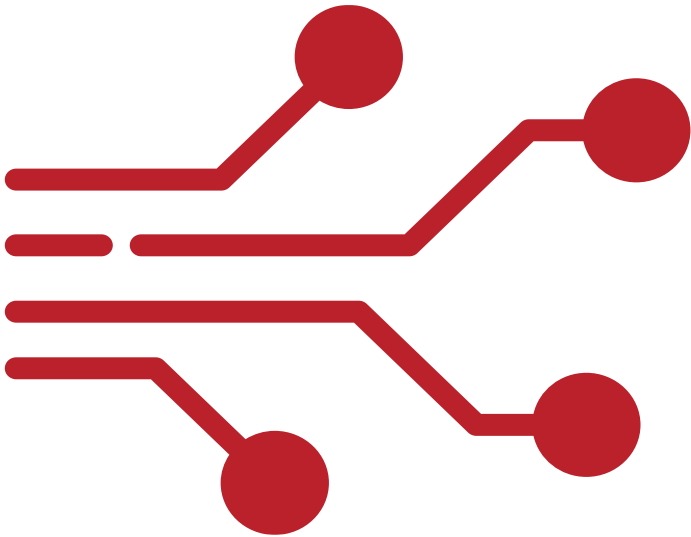
Proces w diagramie DFD

- Reprezentuje operację przetwarzania danych
- Przekształca dane wejściowe w dane wyjściowe
- Powinien mieć jednoznaczną nazwę
- Nazwa powinna opisywać działanie
- Proces nie istnieje bez danych wejściowych lub wyjściowych



Przepływ danych

- Reprezentuje przekazywanie informacji między elementami systemu
- Łączy procesy, magazyny danych i encje zewnętrzne
- Zawsze ma określony kierunek
- Powinien być opisany nazwą przekazywanych danych
- Nie reprezentuje przepływu sterowania



Magazyn danych

- Reprezentuje miejsce przechowywania informacji
- Może odpowiadać bazie danych lub plikowi
- Dane mogą być zapisywane i odczytywane przez procesy
- Nie przetwarza danych samodzielnie
- Przechowuje dane w sposób trwały



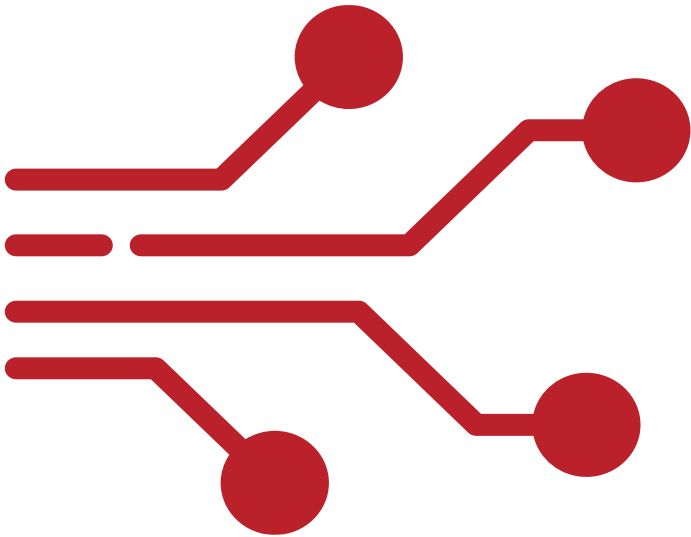
Encja zewnętrzna

- Reprezentuje element znajdujący się poza systemem
- Dostarcza dane do systemu lub odbiera z niego informacje
- Może być użytkownikiem, organizacją lub innym systemem
- Nie jest częścią analizowanego systemu
- Komunikuje się z systemem poprzez przepływy danych



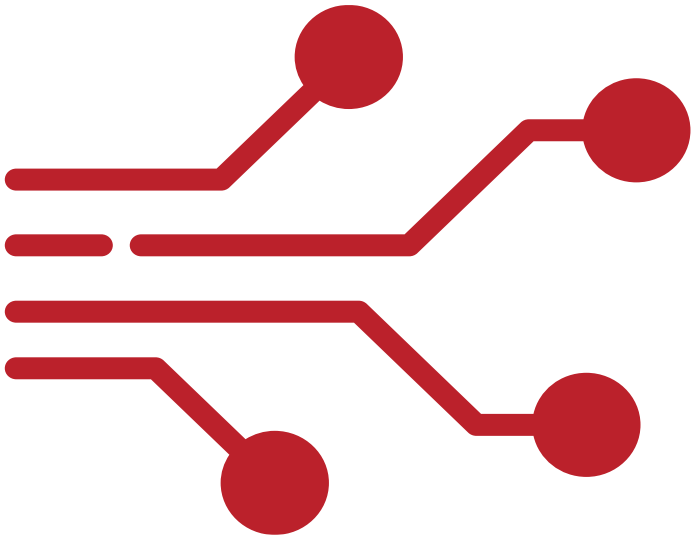
Diagram kontekstowy

- Najwyższy poziom modelu systemu
- Przedstawia system jako jeden proces
- Pokazuje relacje z encjami zewnętrznymi
- Określa granice systemu
- Ułatwia zrozumienie otoczenia systemu



Dekompozycja diagramu DFD

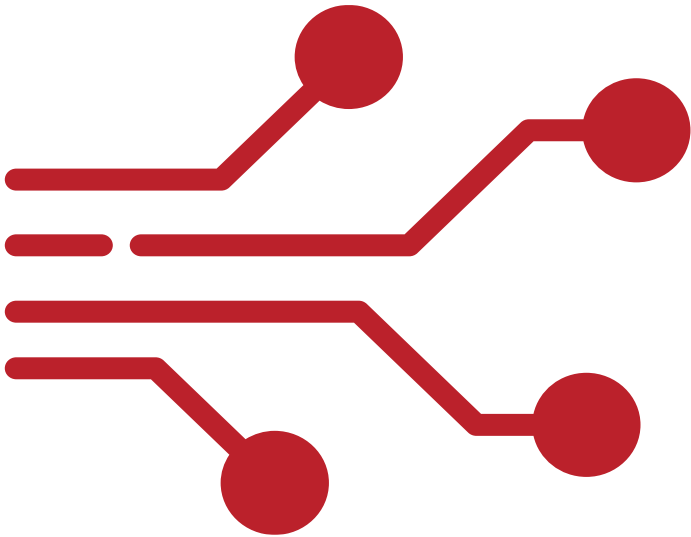
- Podział złożonego procesu na mniejsze procesy
- Przejście od modelu ogólnego do bardziej szczegółowego
- Każdy poziom modelu pokazuje większą szczegółowość
- Zachowanie zgodności przepływów danych
- Ułatwia analizę złożonych systemów



111

Hierarchia diagramów DFD

- Diagram kontekstowy - poziom najwyższy
- Diagram poziomu 0 - główne procesy systemu
- Diagramy niższych poziomów - szczegółowa dekompozycja
- Każdy poziom zwiększa szczegółowość modelu
- Zachowanie spójności między poziomami



112

Zasady poprawnego modelowania DFD

- Każdy proces powinien mieć dane wejściowe i wyjściowe
- Przepływy danych powinny być nazwane
- Nazwy procesów powinny opisywać działanie
- Magazyny danych nie komunikują się bezpośrednio z encjami zewnętrznymi
- Diagram powinien być czytelny i logiczny



Typowe błędy w diagramach DFD

- Proces bez danych wejściowych
- Proces bez danych wyjściowych
- Nienazwane przepływy danych
- Bezpośrednie połączenie encji z magazynem danych
- Nadmierna złożoność diagramu

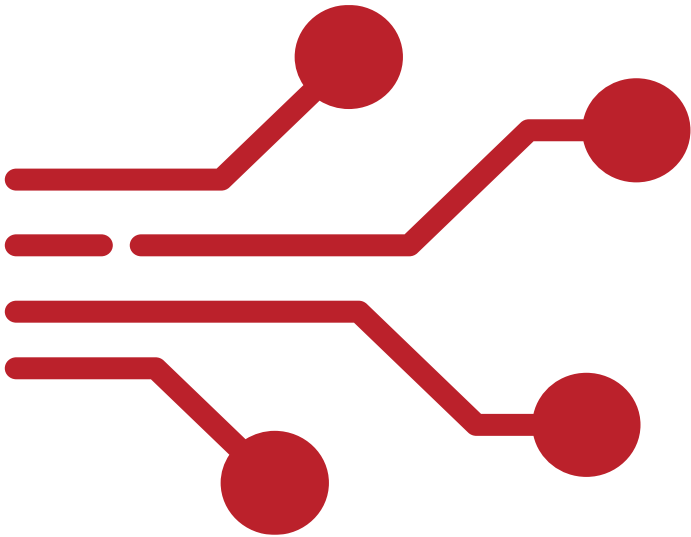
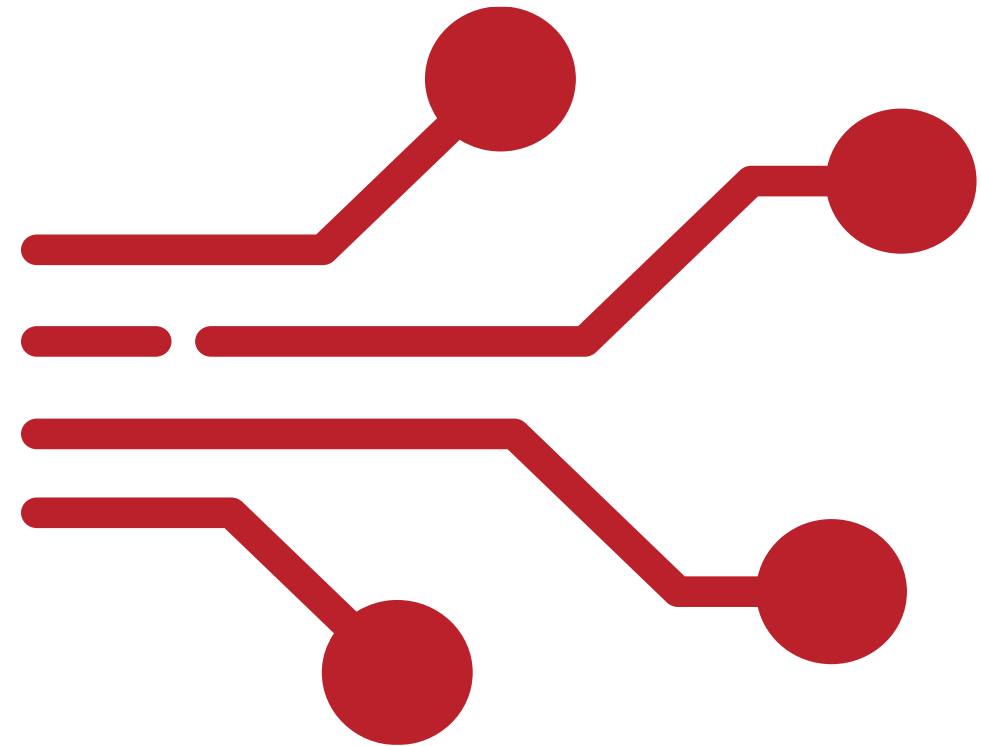


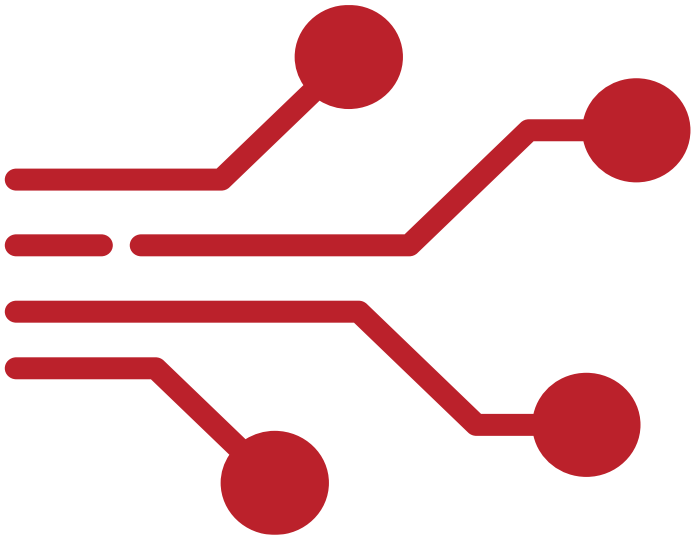
Diagram encji



115

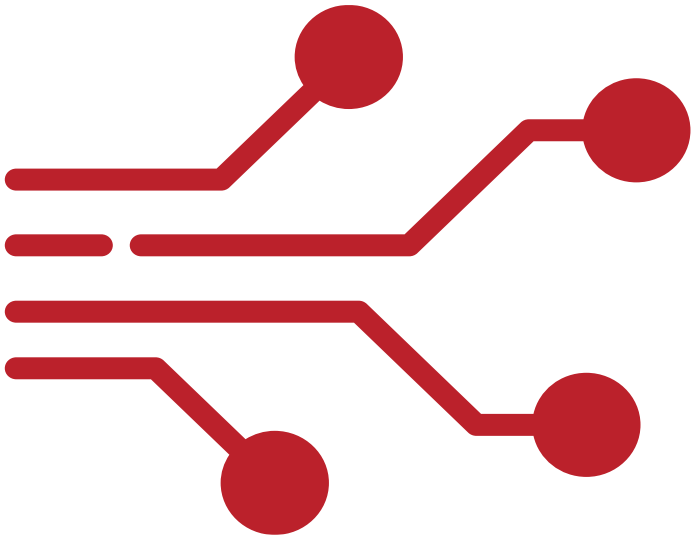
Dlaczego modelujemy dane?

- Dane są podstawą działania systemu
- Wiele procesów korzysta z tych samych informacji
- Konieczne jest uporządkowanie struktury danych
- Model danych pomaga uniknąć redundancji
- Stanowi podstawę projektowania bazy danych



Model encja-związek (ER)

- Model opisujący strukturę danych w systemie
- Przedstawia encje oraz relacje między nimi
- Umożliwia uporządkowanie informacji w systemie
- Stanowi podstawę projektowania bazy danych
- Powszechnie stosowany w analizie systemów



Encja

- Reprezentuje obiekt występujący w systemie
- Opisuje klasę podobnych obiektów
- Posiada zestaw atrybutów
- Każdy obiekt jest instancją encji
- Przykłady: klient, zamówienie, produkt



Atrybut encji

- Opisuje właściwość encji
- Reprezentuje pojedynczą informację o obiekcie
- Każda encja może posiadać wiele atrybutów
- Atrybuty mogą mieć różne typy danych
- Przykłady: imię, numer zamówienia, data zakupu



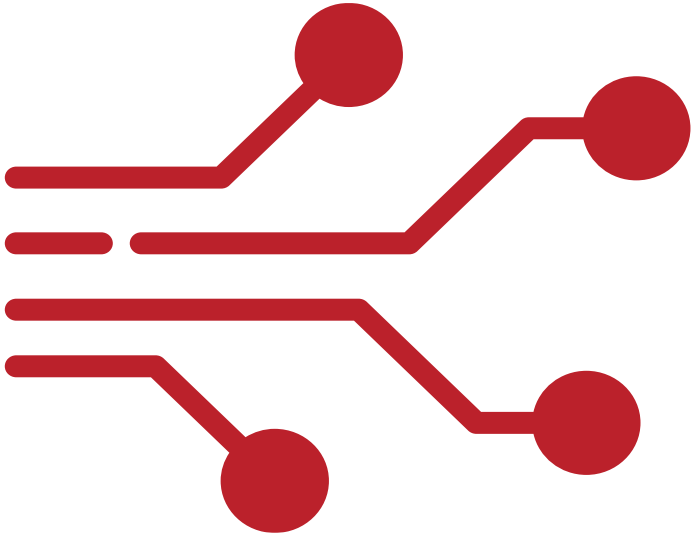
Klucz główny

- Atrybut identyfikujący jednoznacznie obiekt
- Każda encja powinna posiadać klucz główny
- Wartość klucza jest unikalna
- Umożliwia odróżnienie instancji encji
- Przykłady: ID klienta, numer zamówienia



Związki między encjami

- Opisują relacje między obiektami w systemie
- Łączą dwie lub więcej encji
- Reprezentują zależności między danymi
- Mogą mieć własne atrybuty
- Przykład: klient składa zamówienie



Kardynalność relacji

- Określa liczbę powiązań między encjami
- Definiuje, ile obiektów jednej encji może być powiązanych z drugą
- Najczęściej wyróżnia się trzy typy relacji
 - Jeden do jednego (1:1)
 - Jeden do wielu (1:N)
 - Wiele do wielu (N:M)

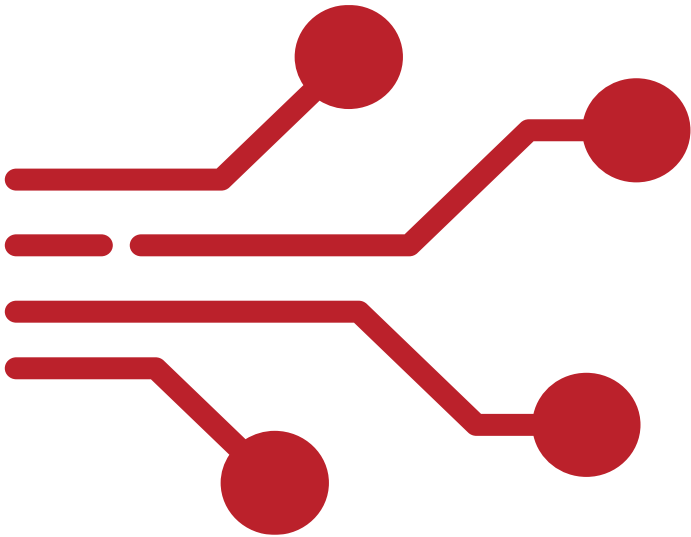
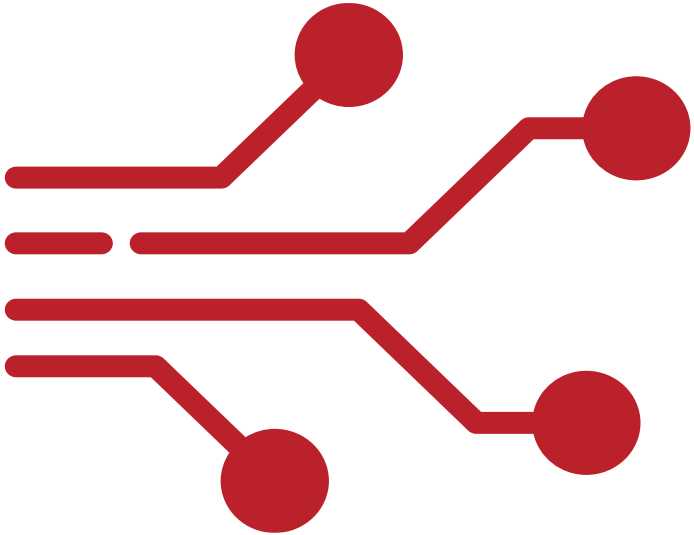
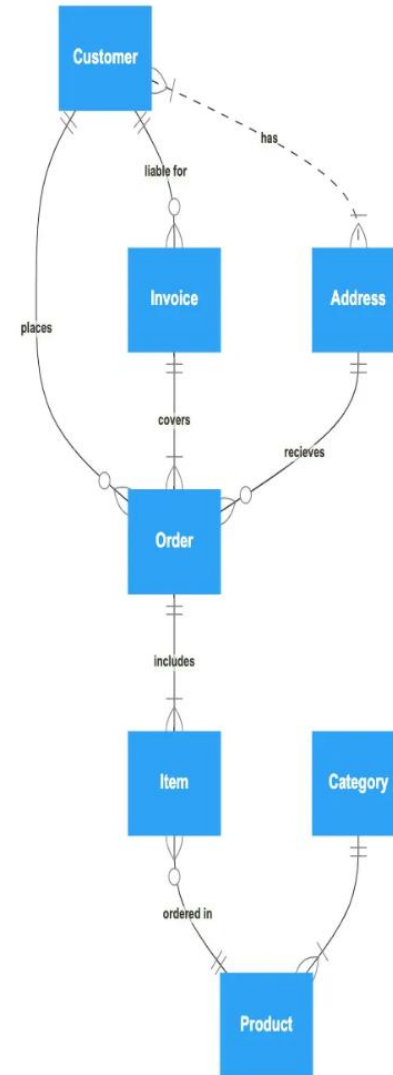


Diagram ER w analizie systemu

- Graficzna reprezentacja modelu danych
- Przedstawia encje, atrybuty i relacje
- Umożliwia wizualizację struktury danych
- Pomaga wykryć niespójności w modelu
- Stanowi podstawę projektowania bazy danych



Przykład

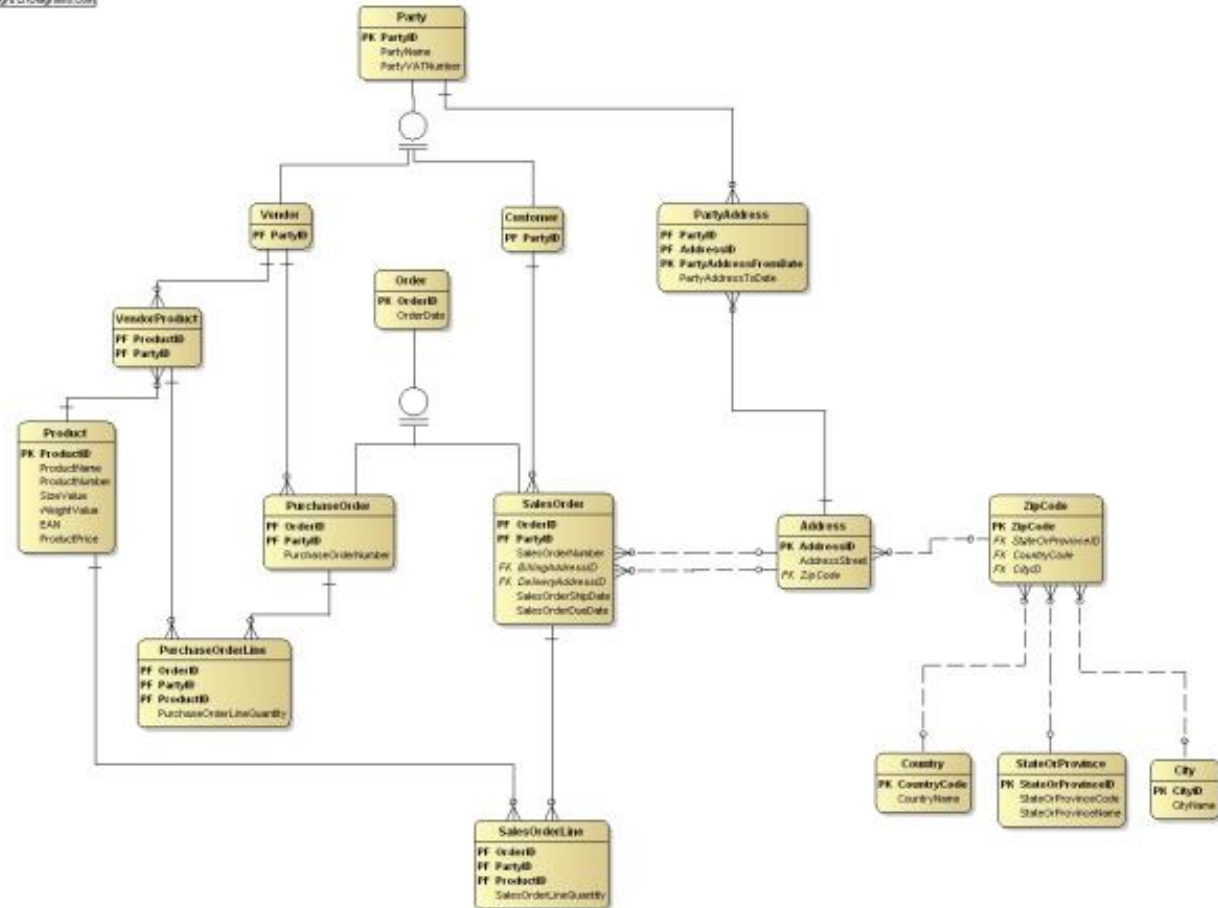


Rys. 3. Źródło: <https://www.gleek.io/templates/er-order-process>

124

Przykład

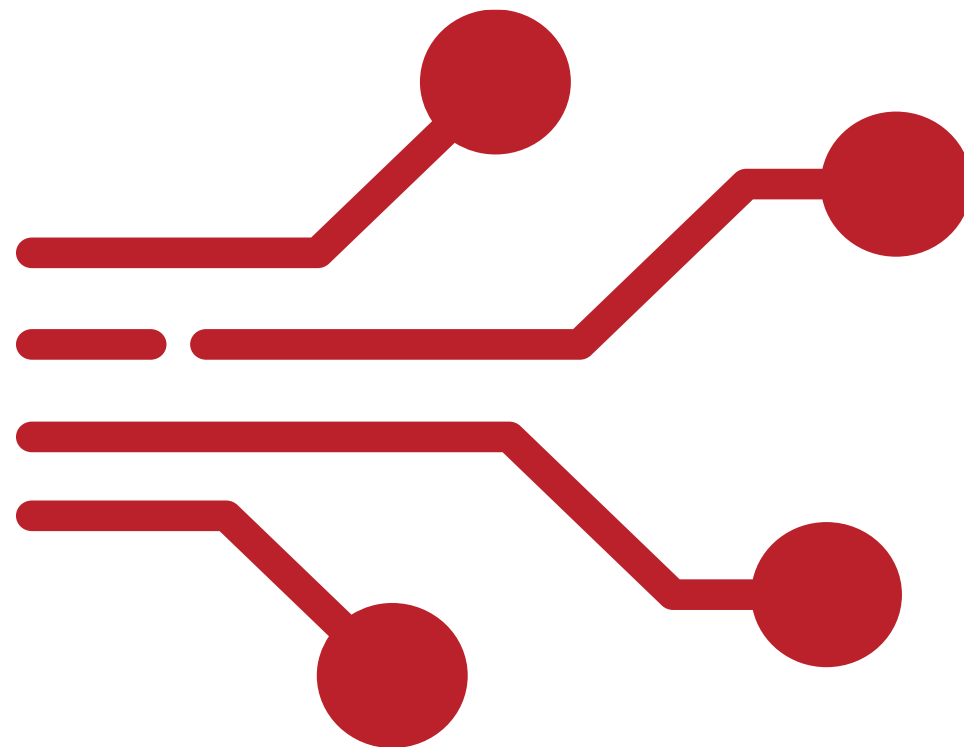
Copyright ERDiagrams.com



Rys. 4. Źródło: <https://www.erdiagrams.com/datamodel-orders-crowsfoot.html>

125

Integracja modeli



126

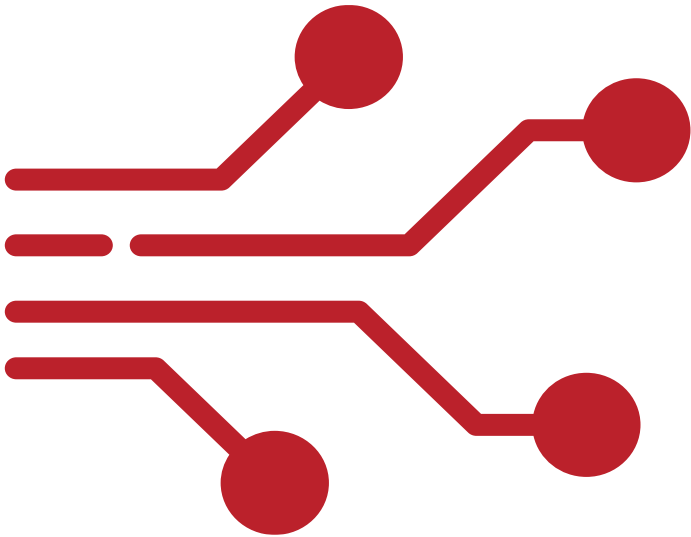
Relacja między DFD a ER

- DFD opisuje procesy przetwarzania danych
- ER opisuje strukturę danych
- Procesy korzystają z danych przechowywanych w encjach
- Modele uzupełniają się w analizie systemu
- Razem tworzą pełniejszy obraz systemu



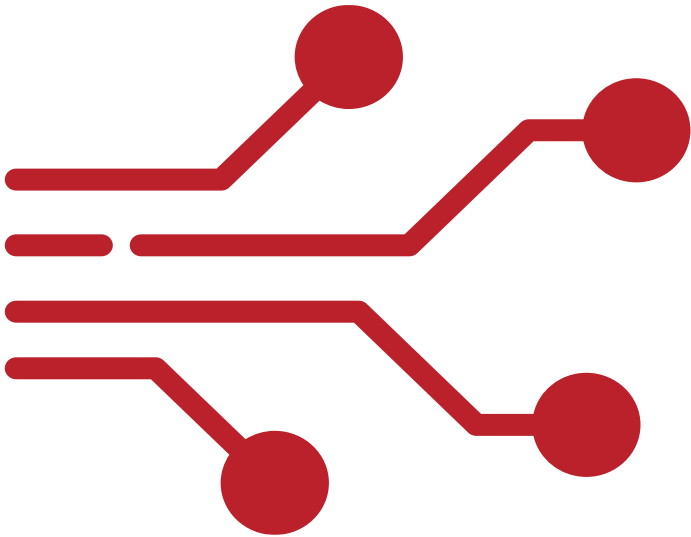
Procesy a dane w modelu systemu

- Procesy przetwarzają dane
- Dane są wejściem i wyjściem procesów
- Procesy mogą zapisywać dane w magazynach
- Procesy mogą odczytywać dane z magazynów
- Dane umożliwiają realizację funkcji systemu



Spójność modeli analitycznych

- Modele procesów i danych powinny być zgodne
- Dane używane w procesach muszą istnieć w modelu danych
- Każdy magazyn danych powinien mieć odpowiednik w modelu ER
- Modele powinny opisywać ten sam zakres systemu
- Spójność modeli zwiększa jakość analizy



Od modelu analitycznego do projektu bazy danych

- Model ER stanowi podstawę projektowania bazy danych
- Encje odpowiadają tabelom
- Atrybuty odpowiadają kolumnom
- Relacje odwzorowywane są za pomocą kluczy obcych
- Kardynalność wpływa na strukturę tabel



Metody strukturalne w praktyce projektowej

- Stosowane w analizie systemów informatycznych
- Szczególnie popularne w starszych projektach
- Wykorzystywane w projektowaniu baz danych
- Pomagają zrozumieć przepływ informacji w systemie
- Stanowią podstawę wielu współczesnych metod



131

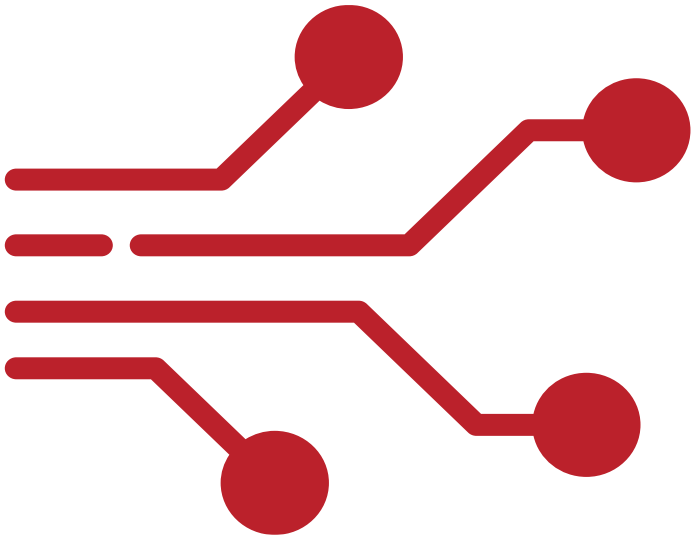
Zalety podejścia strukturalnego

- Przejrzyste modelowanie procesów
- Wyraźne oddzielenie danych i procesów
- Ułatwia analizę złożonych systemów
- Czytelne diagramy dla analityków
- Dobra podstawa projektowania baz danych



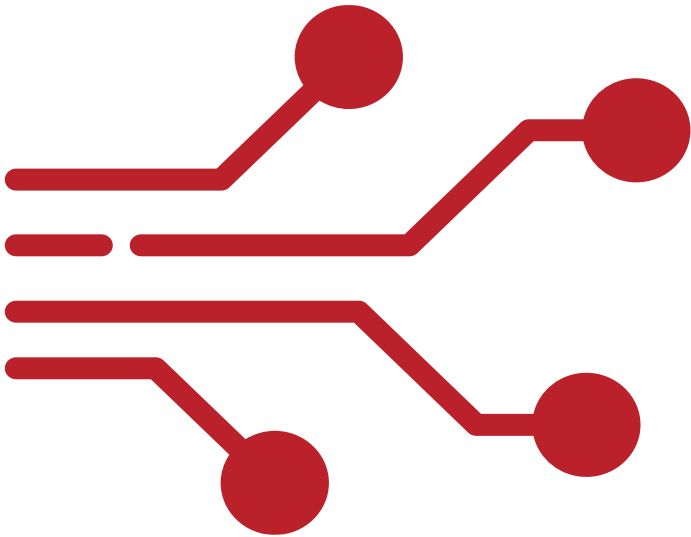
Ograniczenia metod strukturalnych

- Trudność modelowania złożonych zależności
- Rozdzielenie danych i procesów nie zawsze jest naturalne
- Słabe odwzorowanie obiektów świata rzeczywistego
- Mniej przydatne w dużych systemach obiektowych
- Ograniczone wsparcie dla ponownego użycia komponentów



Metody strukturalne a podejście obiektowe

- Metody strukturalne koncentrują się na procesach
- Podejście obiektowe koncentruje się na obiektach
- W metodach strukturalnych dane i procesy są rozdzielone
- W podejściu obiektowym dane i operacje są połączone
- UML stał się standardem modelowania obiektowego



134

Kiedy stosować modele strukturalne?

- W analizie przepływu informacji w systemie
- W projektowaniu baz danych
- W systemach przetwarzania danych
- W analizie istniejących systemów
- W celach edukacyjnych i analitycznych



Współczesne narzędzia modelowania

- Narzędzia do modelowania UML i ER
- Enterprise Architect
- Visual Paradigm
- Lucidchart, Draw.io
- Narzędzia wspierające dokumentację projektową



Formularz opisu wymagania w analizie strukturalnej

- Wymagania opisywane są w ustrukturyzowanej formie
- Każdy proces posiada opis tekstowy
- Formularz uzupełnia diagramy DFD
- Opisuje logikę działania procesu
- Zawiera dane wejściowe i wyjściowe



Typowa struktura formularza opisu procesu

- Nazwa procesu
- Opis działania procesu
- Dane wejściowe
- Dane wyjściowe
- Reguły przetwarzania danych
- Warunki wyjątkowe



Przykład formularza opisu procesu

Nazwa procesu (funkcjonalności): (ew. id) Rejestracja zamówienia

Opis procesu: Proces przyjmuje dane zamówienia od klienta, weryfikuje poprawność danych oraz zapisuje zamówienie w systemie.

Dane wejściowe: Dane klienta, lista produktów, ilość produktów

Dane wyjściowe: Potwierdzenie zamówienia, zapis zamówienia w bazie danych

Reguły przetwarzania: Sprawdzenie poprawności danych klienta, sprawdzenie dostępności produktów, obliczenie wartości zamówienia

Sytuacje wyjątkowe: Brak produktu w magazynie, niepoprawne dane klienta

Dziękuję za uwagę

Wykład nr 4 Temat: Metody obiektowe analizy i projektowania oprogramowania. Diagramy przypadków użycia. Przypadki użycia i ich scenariusze. Diagramy klas i obiektów

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

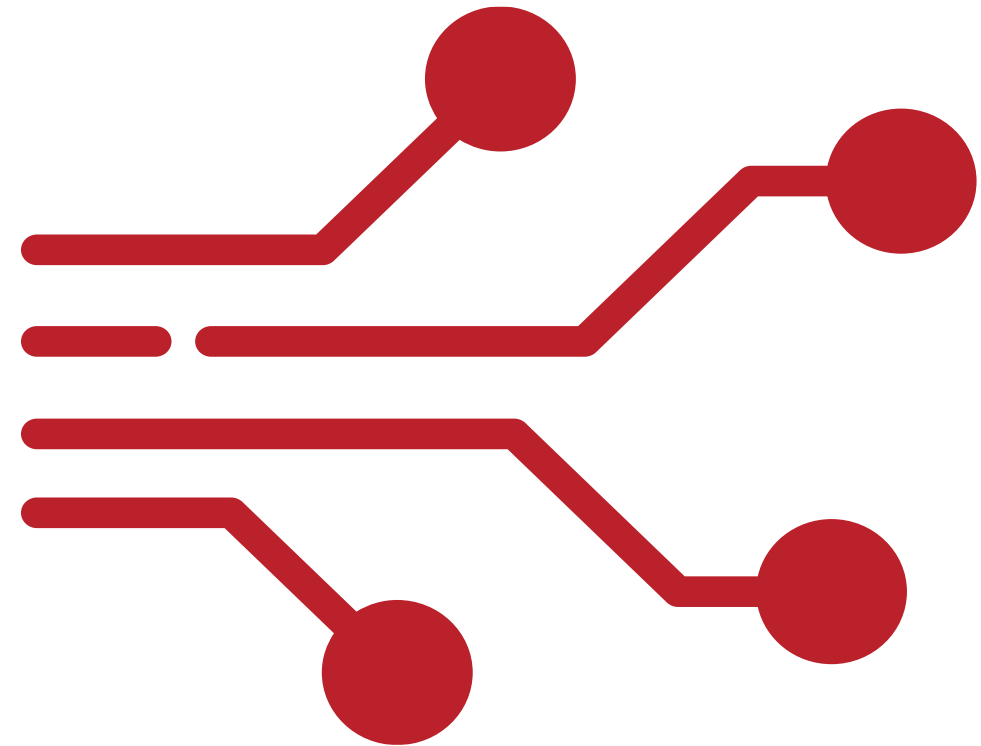
Plan prezentacji

- Wprowadzenie do podejścia obiektowego
- UML
- Diagram przypadków użycia
- Opis przypadku użycia
- Diagram klas
- Diagram obiektów



142

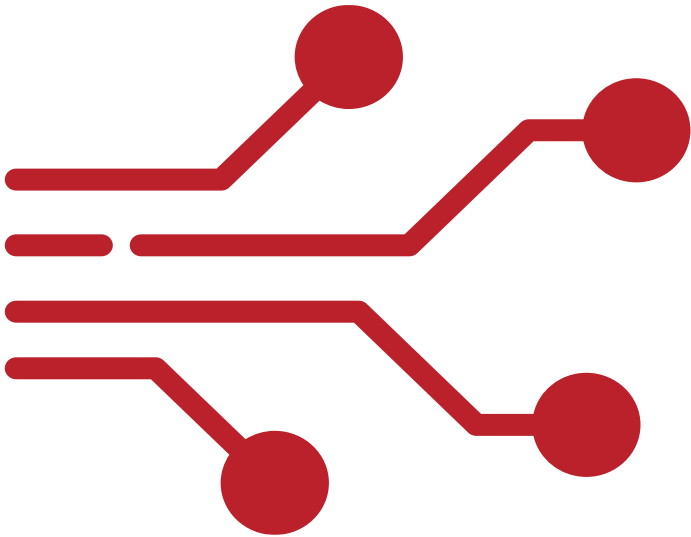
Wprowadzenie do podejścia obiekтового



143

Geneza podejścia obiektowego

- Rosnąca złożoność systemów informatycznych
- Ograniczenia metod strukturalnych
- Potrzeba modelowania obiektów świata rzeczywistego
- Rozwój języków programowania obiektowego
- Integracja danych i operacji w jednym modelu



Idea programowania obiektowego

- System przedstawiany jest jako zbiór obiektów
- Obiekty posiadają dane i operacje
- Obiekty komunikują się poprzez wywołania metod
- Model odzwierciedla elementy świata rzeczywistego
- Ułatwia projektowanie złożonych systemów



Podstawowe pojęcia podejścia obiektowego (przypomnienie)

- Obiekt
- Klasa
- Atrybut
- Metoda



146

Hermetyzacja

- Łączenie danych i operacji w jednym obiekcie
- Ukrywanie szczegółów implementacji
- Dostęp do danych poprzez metody
- Ograniczenie bezpośredniego dostępu do pól obiektu
- Zwiększenie bezpieczeństwa i spójności systemu



Związek generalizacji–specjalizacji (modelujący dziedziczenie)

- Relacja między klasą ogólną a klasą bardziej szczegółową
- Klasa specjalizowana dziedziczy cechy klasy ogólnej
- Umożliwia ponowne wykorzystanie struktury klas
- Pozwala budować hierarchie klas
- Ułatwia organizację modelu systemu



Polimorfizm

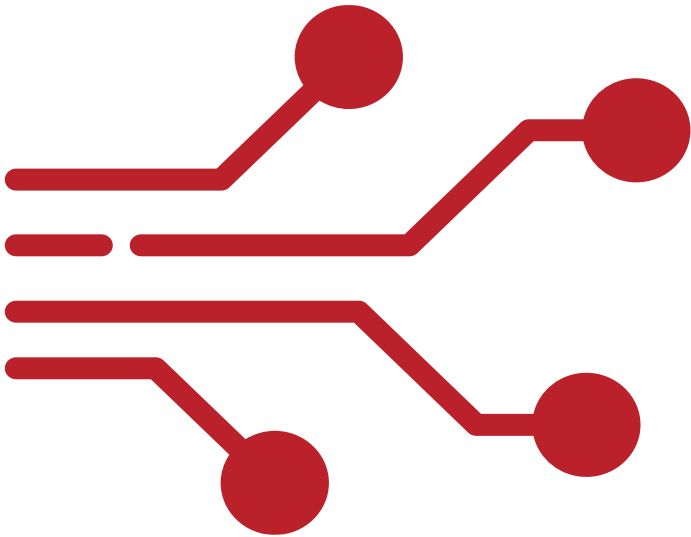
- Możliwość stosowania jednej operacji dla różnych obiektów
- Ta sama metoda może działać różnie w różnych klasach
- Związany ze związkiem generalizacji-specjalizacji
- Umożliwia bardziej elastyczne projektowanie systemów
- Ułatwia rozszerzanie systemu



149

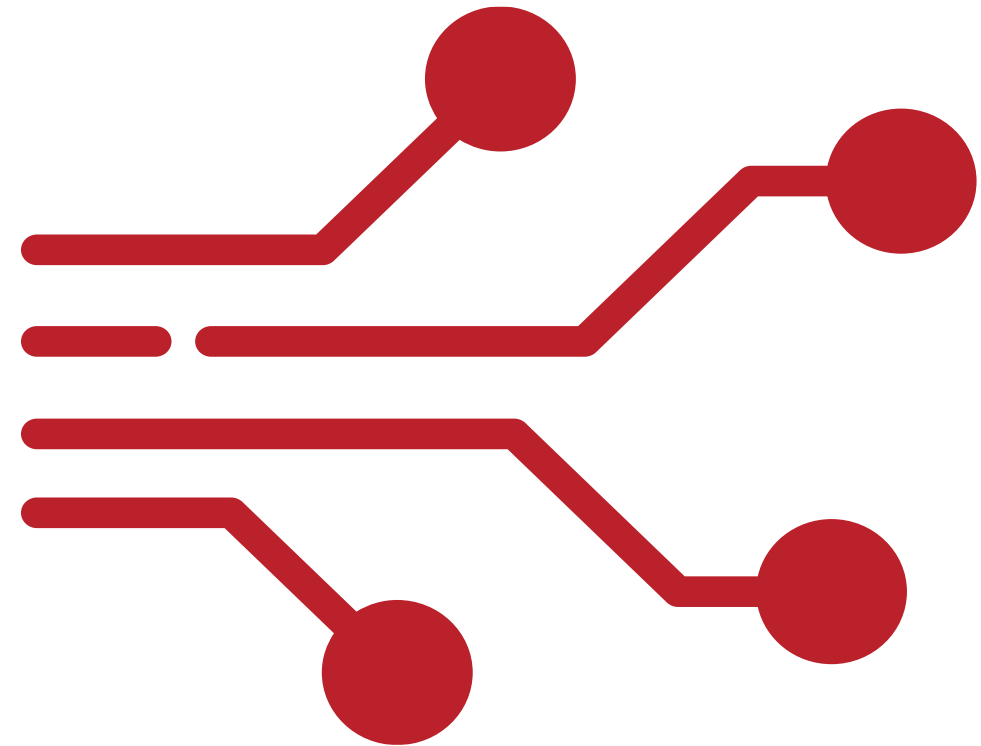
Modelowanie obiektowe w analizie systemu

- System przedstawiany jest jako zbiór współpracujących obiektów
- Obiekty posiadają stan i zachowanie
- Model odzwierciedla strukturę świata rzeczywistego
- Analiza koncentruje się na identyfikacji klas i relacji
- Modele stanowią podstawę projektowania systemu



150

UML



151

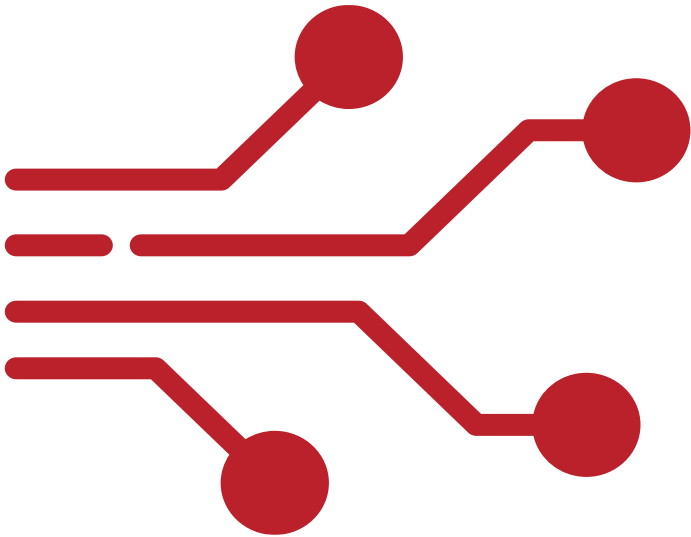
UML (Unified Modeling Language)

- Standardowy język modelowania systemów informatycznych
- Powstał w latach 90. XX wieku
- Ujednolica różne metody modelowania obiektowego
- Umożliwia graficzne przedstawianie systemów
- Powszechnie stosowany w analizie i projektowaniu oprogramowania



Rodzaje diagramów UML

- Diagramy strukturalne: klas, obiektów, pakietów, komponentów, wdrożenia, struktury złożonej, profili
- Diagramy behawioralne: przypadków użycia, aktywności, stanów (maszyny stanowej)
 - Diagramy interakcji: sekwencji, komunikacji, przeglądu interakcji, czasowy (harmonogramowania)



Diagramy strukturalne a diagramy behawioralne

- Diagramy strukturalne
 - Opisują strukturę systemu
 - Pokazują elementy systemu i ich relacje
 - Przedstawiają statyczny model systemu
- Diagramy behawioralne
 - Opisują działanie systemu
 - Pokazują przebieg procesów i interakcje
 - Przedstawiają dynamiczny model systemu

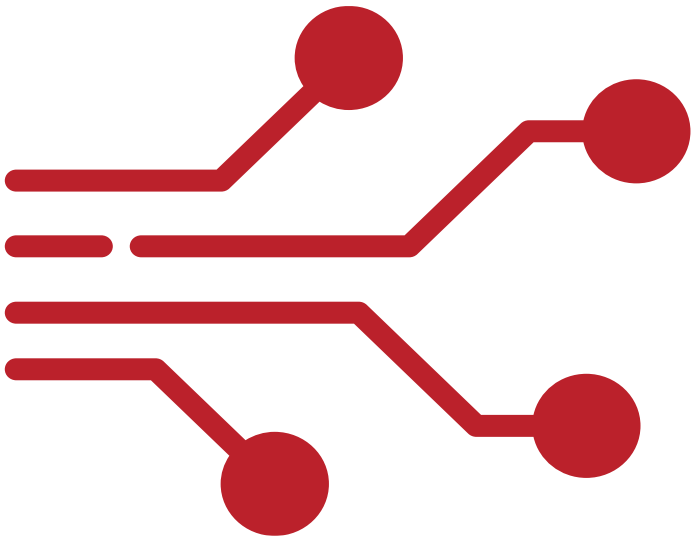
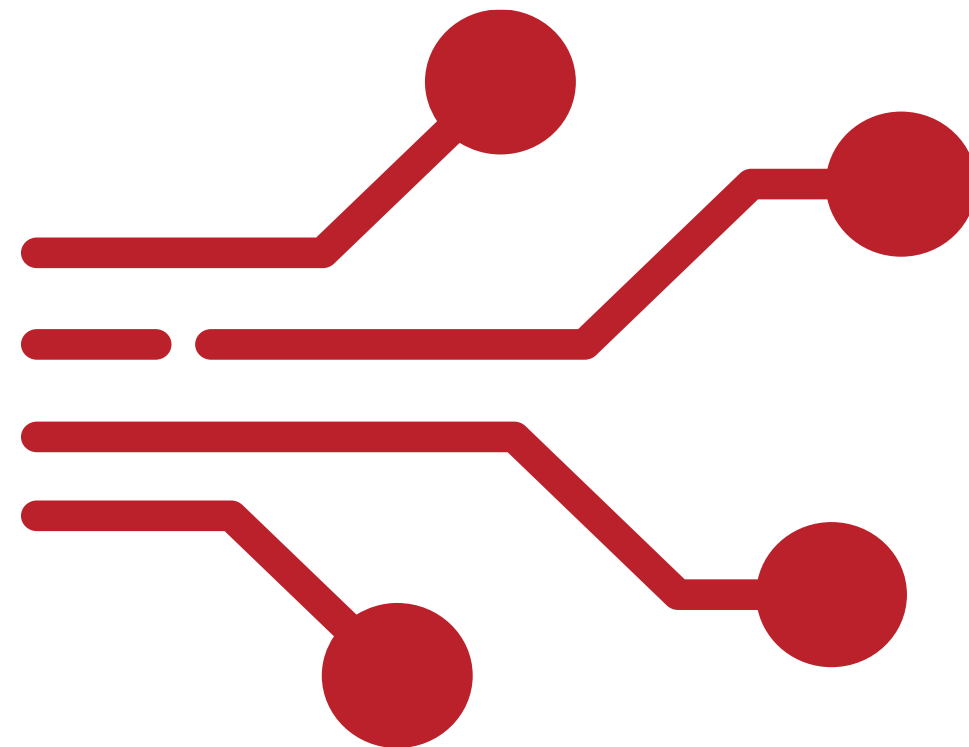


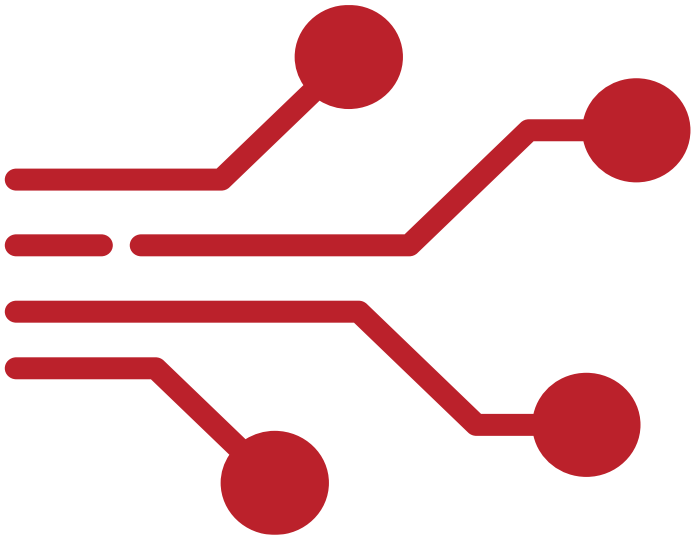
Diagram przypadków użycia



155

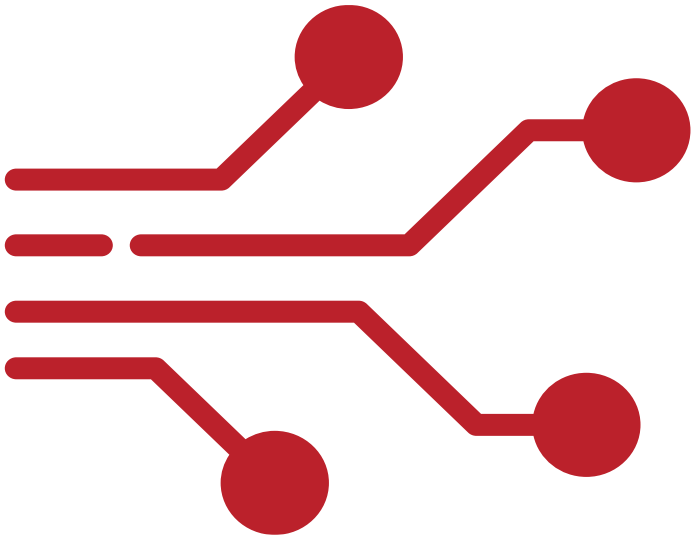
Diagram przypadków użycia

- Przedstawia funkcjonalność systemu
- Pokazuje system z perspektywy użytkownika
- Identyfikuje interakcje między użytkownikiem a systemem
- Pomaga zrozumieć zakres systemu
- Jest często wykorzystywany w analizie wymagań



Elementy diagramu przypadków użycia

- Aktor
- Przypadek użycia
- Granica systemu
- Relacje między przypadkami użycia
- Relacje między aktorem a przypadkiem użycia



Aktor

- Reprezentuje podmiot korzystający z systemu
- Znajduje się poza granicą systemu
- Może być osobą, systemem lub urządzeniem
- Wchodzi w interakcję z przypadkami użycia
- Reprezentuje rolę, a nie konkretną osobę



Przypadek użycia

- Opisuje funkcjonalność systemu
- Przedstawia działanie systemu z perspektywy użytkownika
- Reprezentuje interakcję między aktorem a systemem
- Ma określony cel biznesowy
- Jest przedstawiany na diagramie jako elipsa



Relacje między przypadkami użycia

- Relacja include
- Relacja extend
- Związek generalizacji-specjalizacji (uogólnienia)
- Umożliwiają modelowanie zależności między funkcjami systemu



Relacja include

- Oznacza włączenie jednego przypadku użycia w inny
- Wspólny fragment funkcjonalności systemu
- Używana do eliminacji powtarzających się elementów
- Przypadek użycia bazowy zawsze korzysta z przypadku include
- Na diagramie oznaczana strzałką z napisem «include»



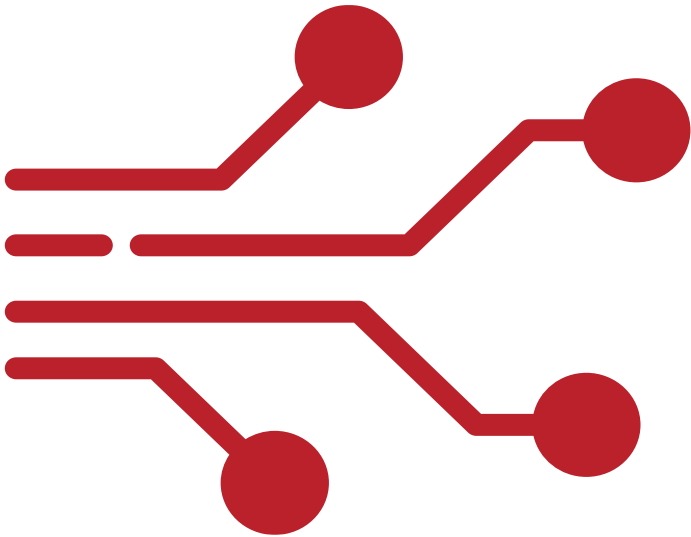
Relacja extend

- Oznacza rozszerzenie działania przypadku użycia
- Wykonywana tylko w określonych warunkach
- Dodaje opcjonalną funkcjonalność
- Przypadek użycia bazowy może działać samodzielnie
- Na diagramie oznaczana strzałką z napisem «extend»

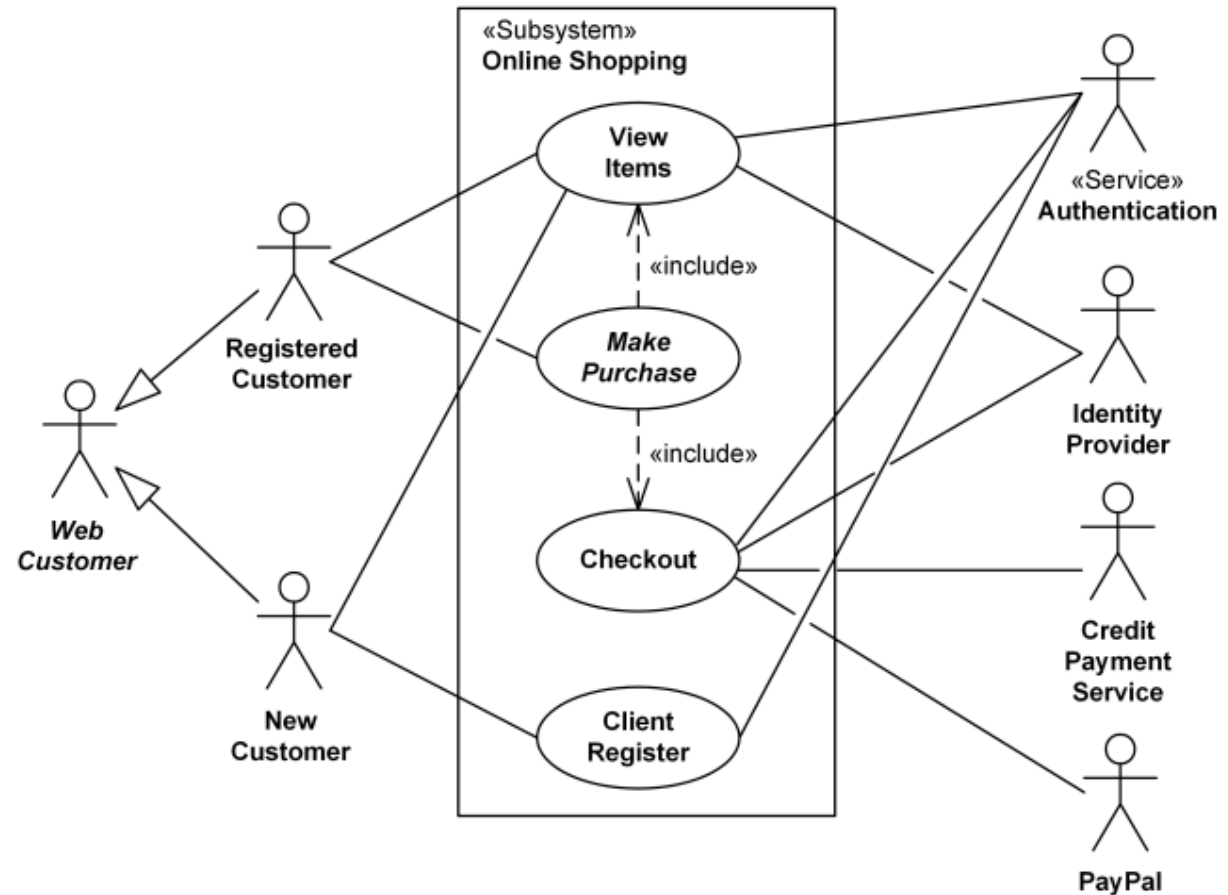


Granica systemu

- Określa zakres systemu
- Oddziela system od jego otoczenia
- Przypadki użycia znajdują się wewnątrz granicy
- Aktorzy znajdują się poza granicą systemu
- Pomaga zrozumieć, co należy do systemu

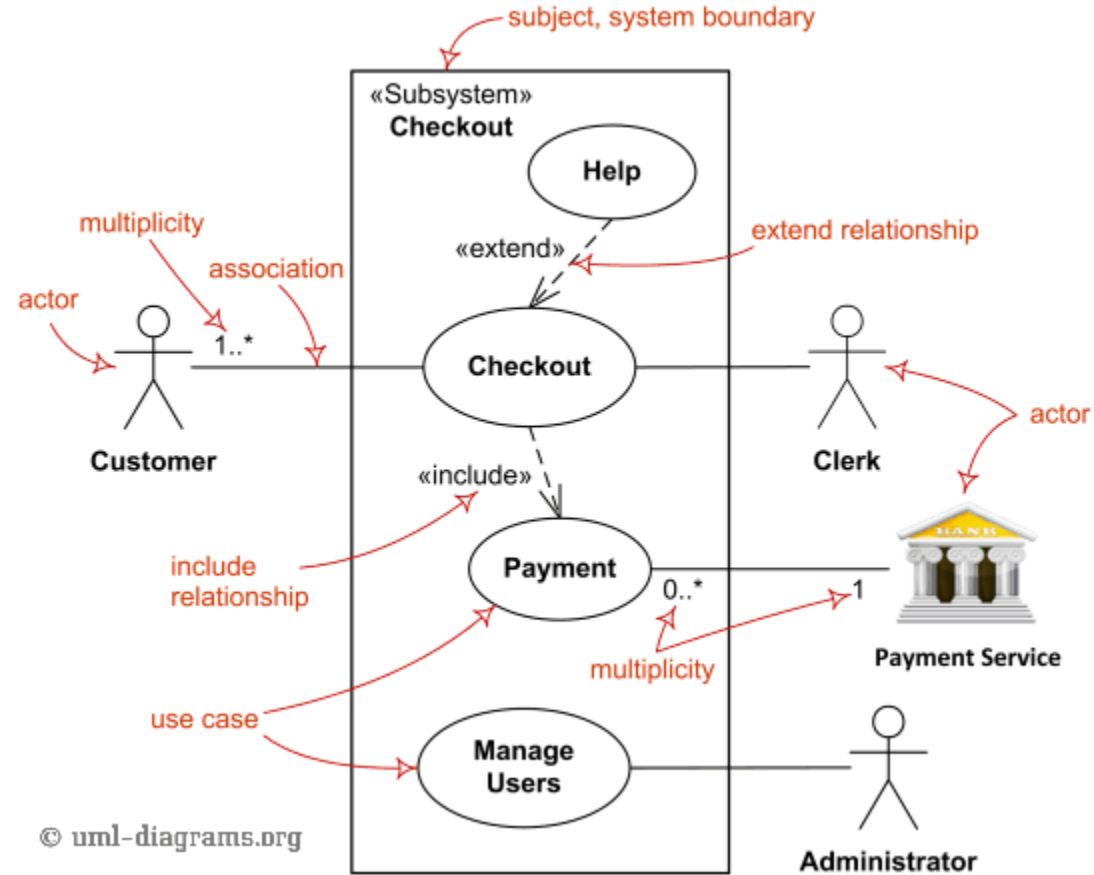


Przykład



Rys. 5. Źródło: <https://www.uml-diagrams.org/examples/online-shopping-use-case-diagram-example.html>

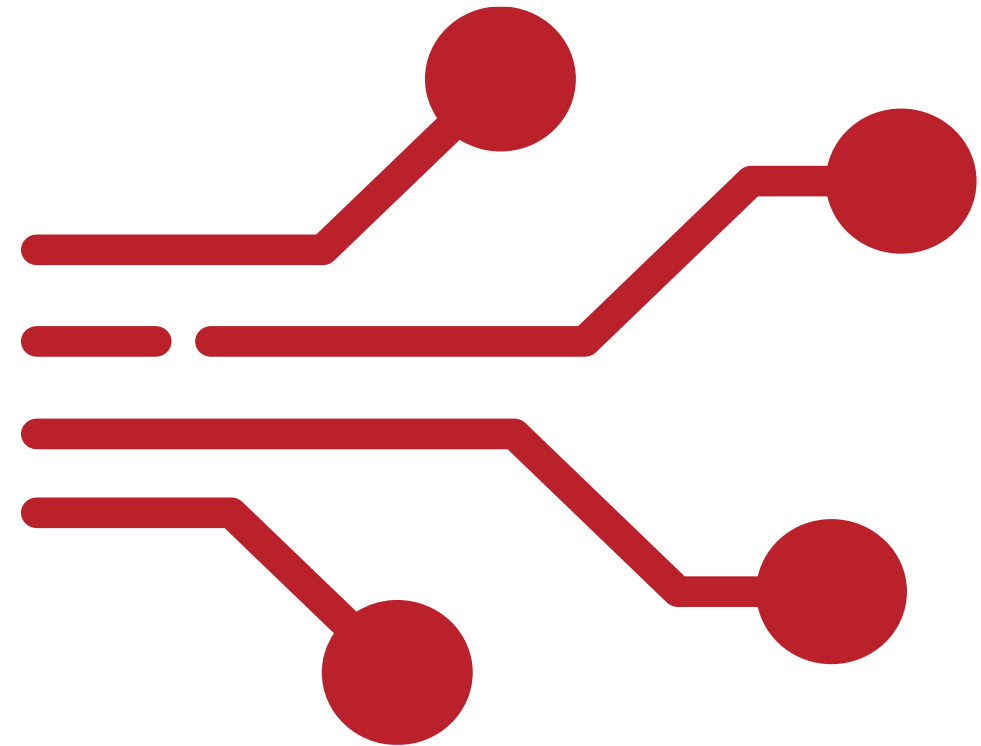
Przykład



Rys. 6. Źródło: <https://www.uml-diagrams.org/use-case-diagrams.html>

165

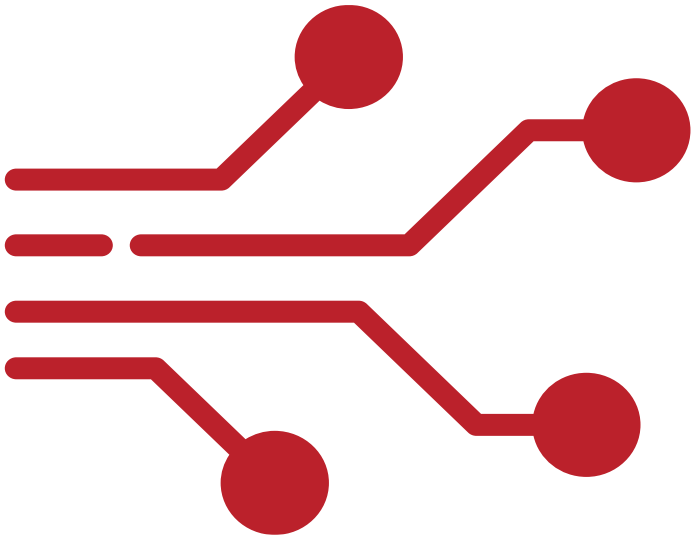
Opis przypadku użycia



166

Dlaczego opisujemy przypadki użycia

- Diagram pokazuje jedynie ogólną funkcjonalność systemu
- Konieczne jest opisanie szczegółowego przebiegu działania
- Opis pozwala zrozumieć logikę funkcji systemu
- Ułatwia komunikację z interesariuszami
- Stanowi podstawę projektowania i implementacji



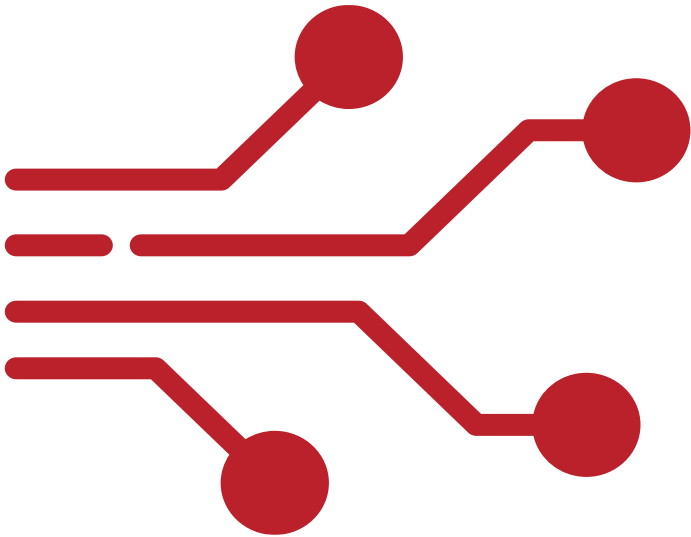
Struktura opisu przypadku użycia

- Nazwa przypadku użycia
- Aktorzy
- Krótki opis funkcji
- Warunki początkowe
- Scenariusz podstawowy
- Scenariusze alternatywne
- Warunki końcowe



Scenariusz podstawowy

- Opisuje typowy przebieg działania systemu
- Przedstawia standardową interakcję aktora z systemem
- Składa się z uporządkowanej sekwencji kroków
- Każdy krok opisuje działanie aktora lub systemu
- Nie uwzględnia sytuacji wyjątkowych



Scenariusze alternatywne

- Opisują inne możliwe przebiegi działania systemu
- Uwzględniają sytuacje wyjątkowe
- Mogą być uruchamiane w określonych warunkach
- Rozszerzają scenariusz podstawowy
- Zwiększają kompletność opisu wymagań



170

Warunki początkowe i końcowe

- Warunki początkowe określają stan systemu przed rozpoczęciem przypadku użycia
- Muszą być spełnione, aby przypadek użycia mógł zostać uruchomiony
- Warunki końcowe opisują stan systemu po zakończeniu działania
- Określają rezultat realizacji funkcjonalności
- Ułatwiają weryfikację poprawności działania systemu



Przykład opisu przypadku użycia

Nazwa: Logowanie użytkownika

Aktor: Użytkownik systemu

Warunki początkowe: Użytkownik posiada konto w systemie

Scenariusz podstawowy:

1. Użytkownik wprowadza login i hasło
2. System weryfikuje dane
3. System umożliwia dostęp do systemu

Scenariusz alternatywny: Podanie niepoprawnego hasła

Warunki końcowe: Użytkownik uzyskuje dostęp do systemu

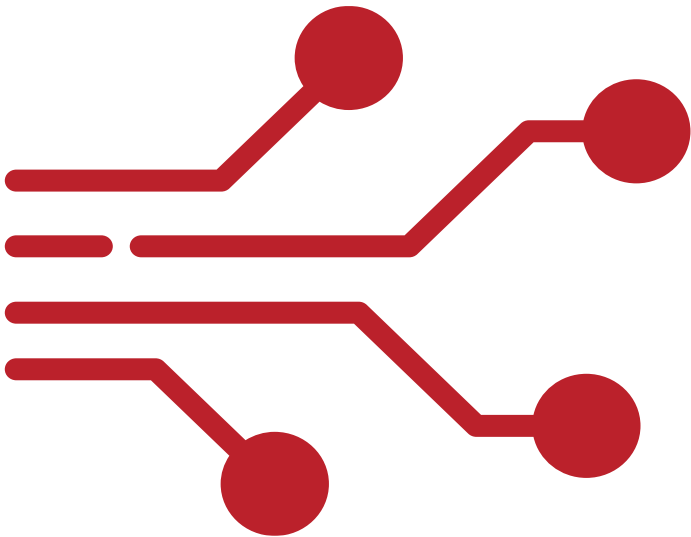
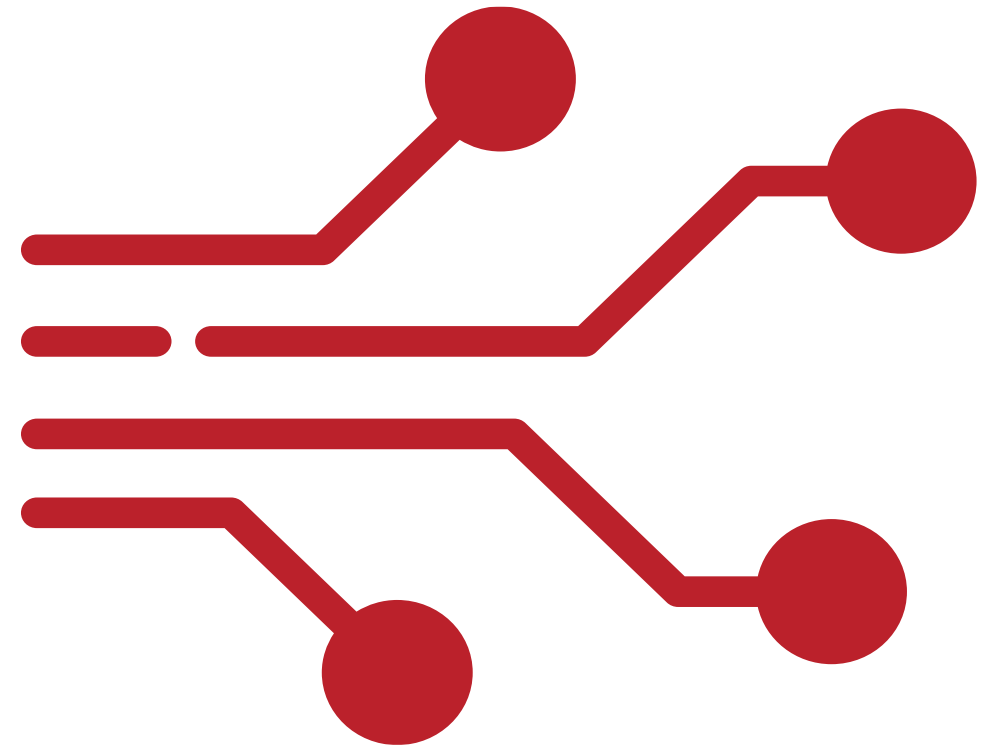


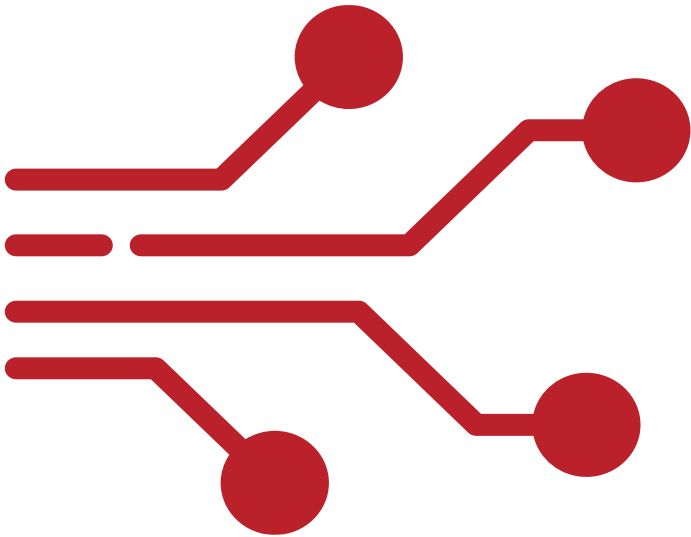
Diagram klas



173

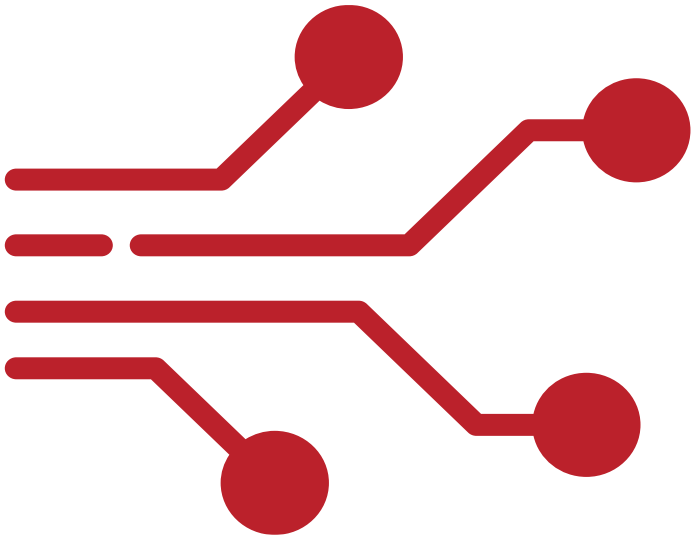
Diagram klas

- Przedstawia strukturę systemu w podejściu obiektowym
- Pokazuje klasy oraz relacje między nimi
- Opisuje statyczny model systemu
- Klasy posiadają atrybuty i operacje
- Jest jednym z najważniejszych diagramów UML



Klasa

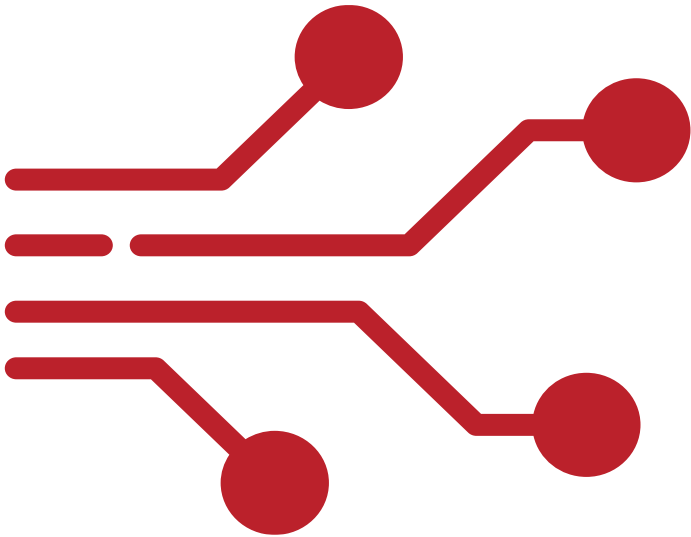
- Reprezentuje typ obiektów w systemie
- Określa strukturę oraz zachowanie obiektów
- Definiuje atrybuty i operacje
- Obiekty są instancjami klas
- Przedstawiana na diagramie jako prostokąt podzielony na sekcje



175

Atrybuty i operacje klasy

- **Atrybuty**
 - Opisują właściwości obiektów
 - Reprezentują dane przechowywane w klasie
 - Mogą mieć określony typ danych
 - Mogą posiadać poziomy dostęp
- **Operacje**
 - Określają działania wykonywane przez obiekty
 - Reprezentują metody klasy
 - Operują na danych obiektu
 - Definiują zachowanie klasy



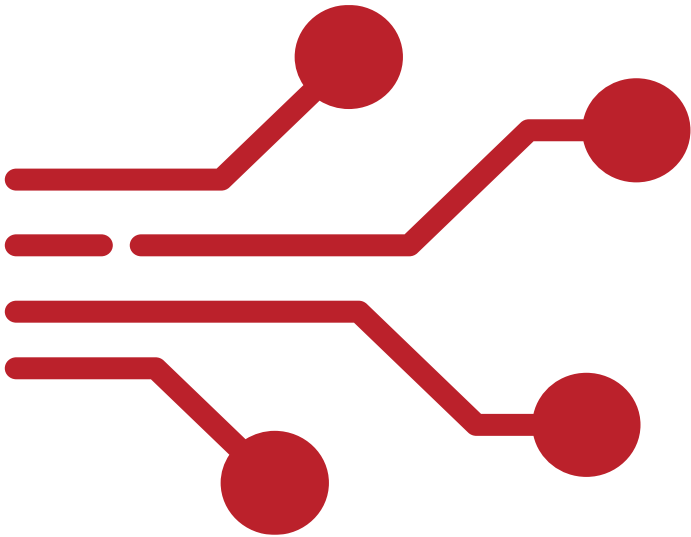
Relacje między klasami

- Asocjacja
- Związek generalizacji-specjalizacji
- Agregacja
- Kompozycja
- Zależność



Asocjacja

- Podstawowy typ relacji między klasami
- Oznacza powiązanie między obiektami
- Przedstawiana jako linia łącząca klasy
- Może posiadać nazwę relacji
- Może zawierać krotność (liczebność)



Związek generalizacji-specjalizacji (uogólnienia)

- Relacja między klasą ogólną a klasą bardziej szczegółową
- Klasa specjalizowana dziedziczy cechy klasy ogólnej
- Umożliwia budowanie hierarchii klas
- Pozwala ponownie wykorzystywać wspólne elementy modelu
- Przedstawiany jako linia zakończona pustym trójkątem



179

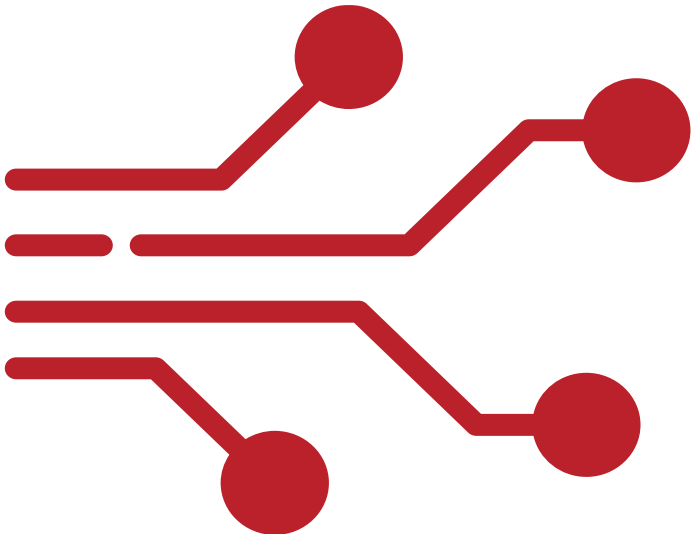
Agregacja i kompozycja

- Agregacja

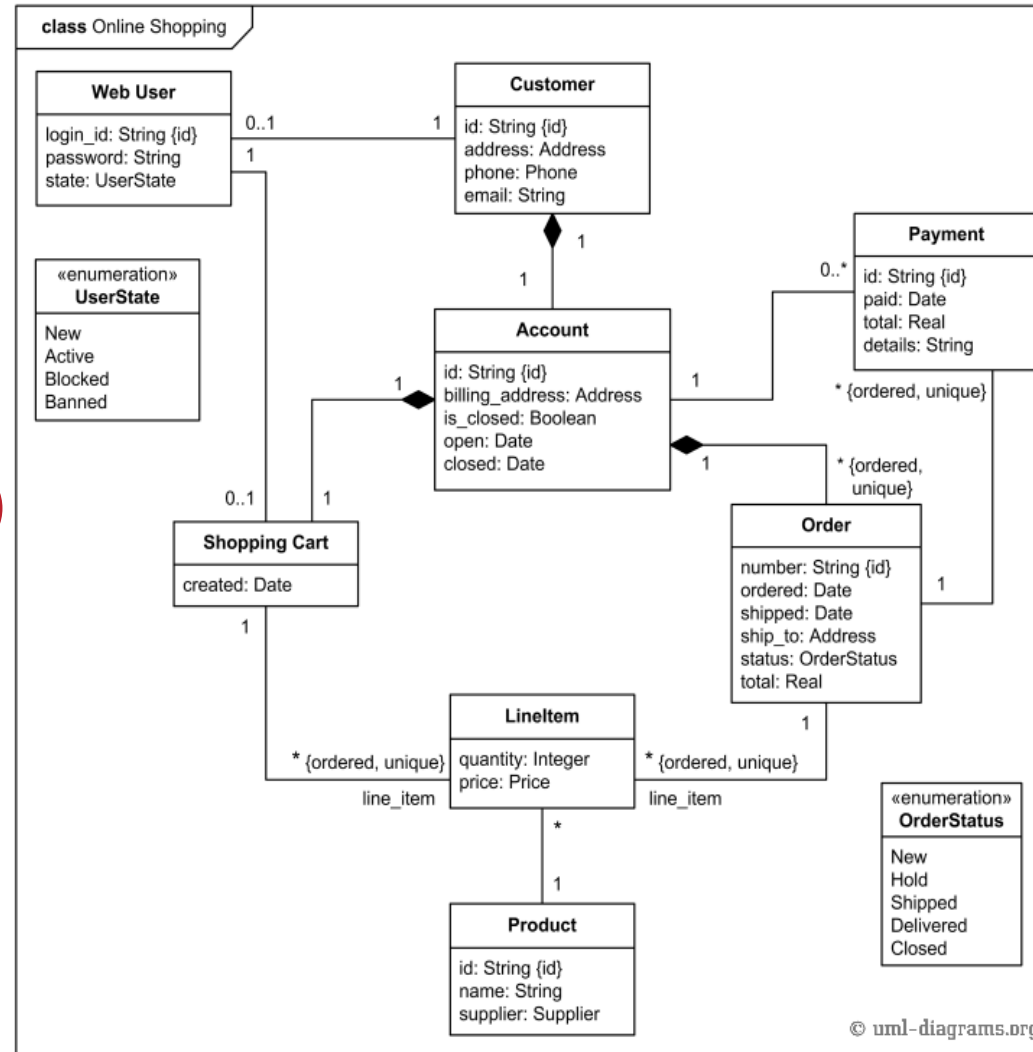
- Relacja typu część-całość
- Obiekty części mogą istnieć niezależnie
- Przedstawiana pustym rombem

- Kompozycja

- Silna relacja typu część-całość
- Obiekty części nie istnieją bez całości
- Przedstawiana wypełnionym rombem

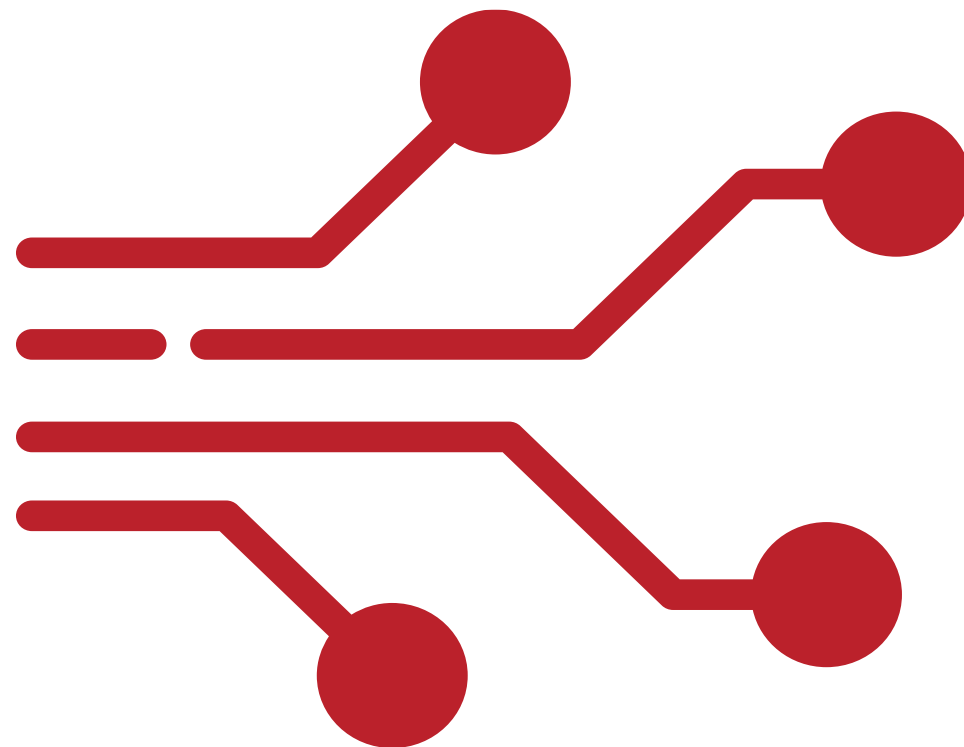


Przykład



Rys. 7. Źródło: <https://www.uml-diagrams.org/examples/online-shopping-domain-uml-diagram-example.html>

Diagram obiektów



182

Diagram obiektów

- Przedstawia konkretne instancje klas
- Pokazuje stan systemu w określonym momencie
- Obiekty są instancjami klas
- Pokazuje powiązania między obiektami
- Jest szczególnym przypadkiem diagramu klas

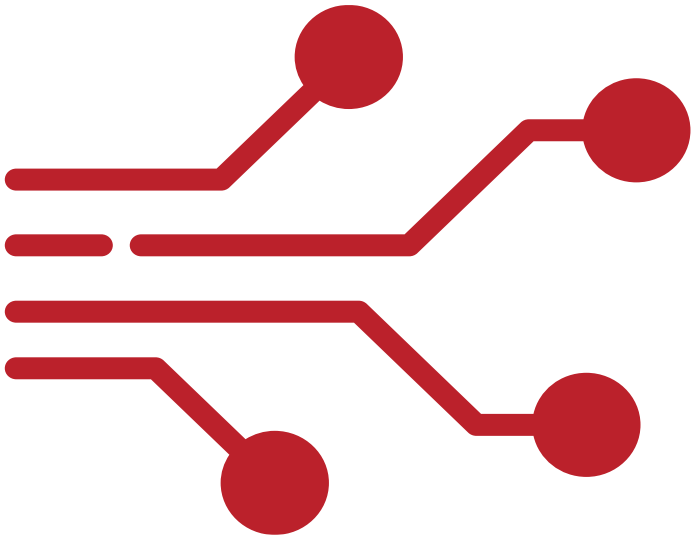


Diagram klas a diagram obiektów

- Diagram klas
 - Przedstawia typy obiektów
 - Pokazuje ogólną strukturę systemu
 - Opisuje klasy, atrybuty i operacje
 - Jest modelem statycznym
- Diagram obiektów
 - Przedstawia konkretne obiekty
 - Pokazuje instancje klas
 - Przedstawia stan systemu w danym momencie
 - Jest przykładem realizacji diagramu klas

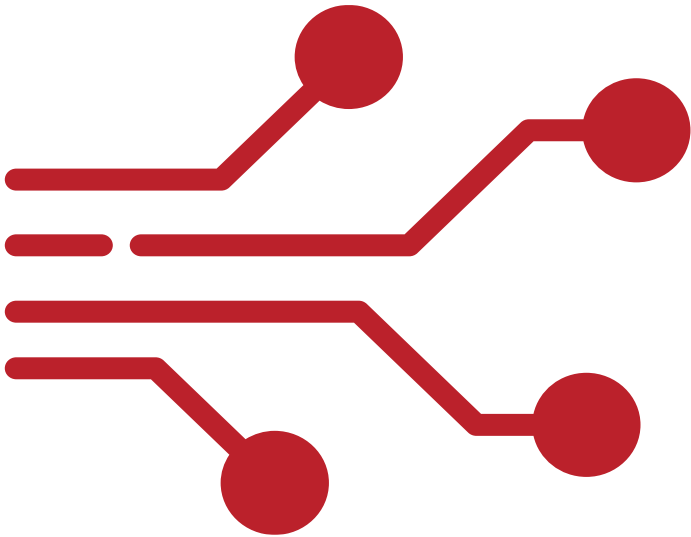
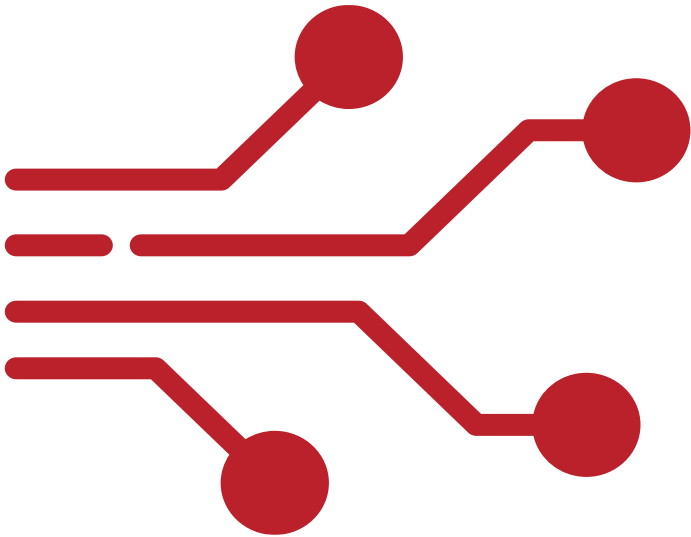
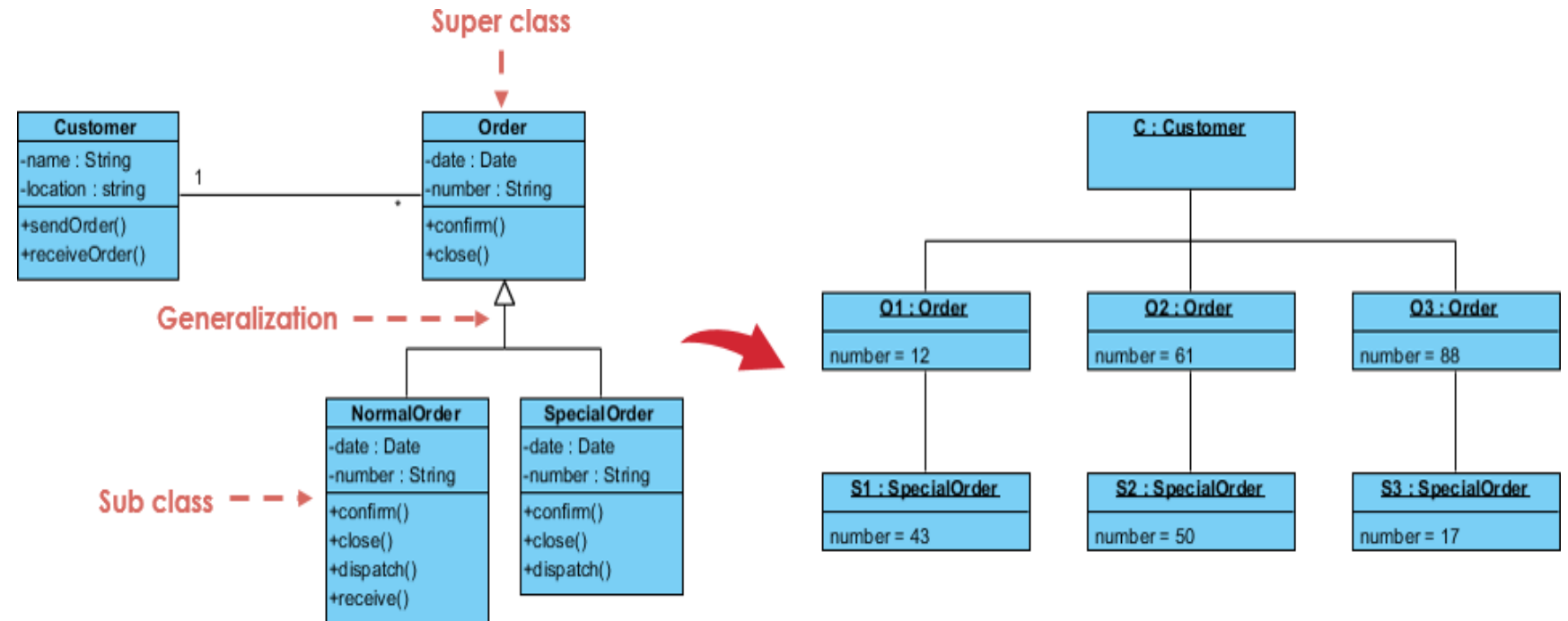


Diagram klas a diagram obiektów

- Diagram klas
 - Przedstawia typy obiektów
 - Pokazuje ogólną strukturę systemu
 - Opisuje klasy, atrybuty i operacje
 - Jest modelem statycznym
- Diagram obiektów
 - Przedstawia konkretne obiekty
 - Pokazuje instancje klas
 - Przedstawia stan systemu w danym momencie
 - Jest przykładem realizacji diagramu klas

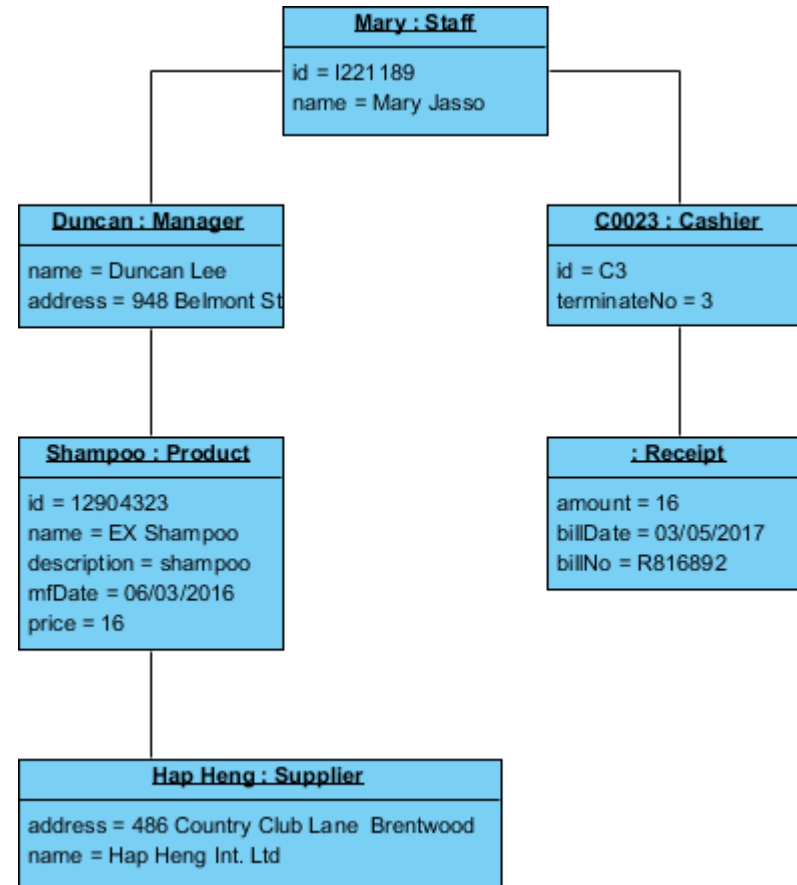


Przykład



Rys. 8. Źródło: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-object-diagram/>

Przykład



Rys. 9. Źródło: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-object-diagram/>

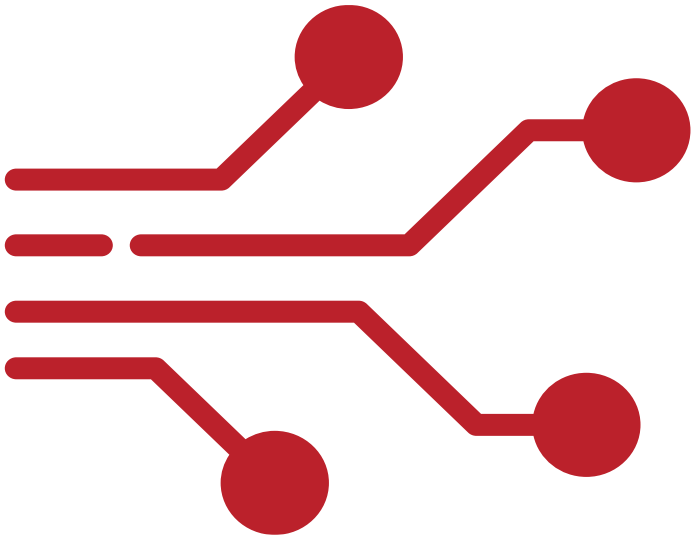
Zastosowanie metod obiektowych

- Analiza wymagań systemu
- Modelowanie funkcjonalności systemu
- Projektowanie struktury systemu
- Tworzenie modeli architektury oprogramowania
- Podstawa implementacji w językach obiektowych



Metody obiektowe a metody strukturalne

- Metody strukturalne
 - Koncentrują się na procesach
 - Oddzielają dane od operacji
 - Wykorzystują diagramy DFD i modele danych
- Metody obiektowe
 - Koncentrują się na obiektach
 - Łączą dane i operacje w jednej strukturze
 - Wykorzystują UML i diagramy klas



Business Process Model and Notation

- BPMN
- Notacja i Model Procesu Biznesowego
- Graficzna notacja służąca do opisywania procesów biznesowych
- Umożliwia tworzenie czytelnych diagramów (za pomocą zdarzeń, zadań, bramek, ułatwiających komunikację między biznesem a IT)



Podsumowanie wykładu

- Podejście obiektowe modeluje system jako zbiór obiektów
- UML jest standardowym językiem modelowania systemów
- Diagram przypadków użycia opisuje funkcjonalność systemu
- Diagram klas przedstawia strukturę systemu
- Diagram obiektów pokazuje konkretne instancje klas



Dziękuję za uwagę

Wykład nr 5

Temat: Jakość kodu. Przeglądy kodu

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

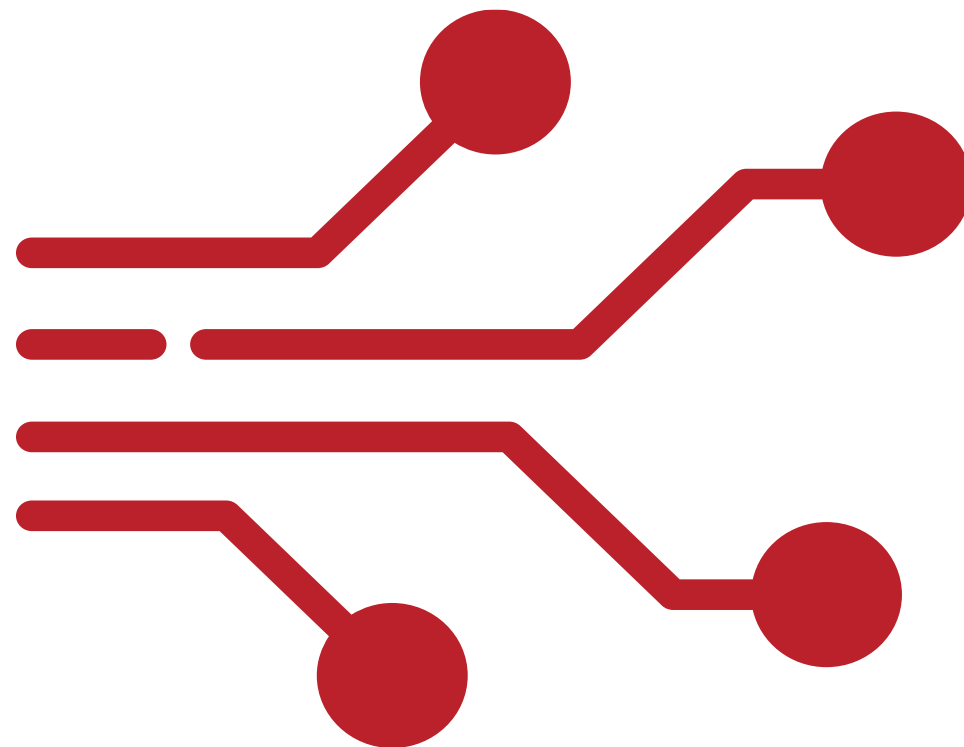
Plan prezentacji

- Jakość kodu
- Przeglądy kodu i dobre praktyki



194

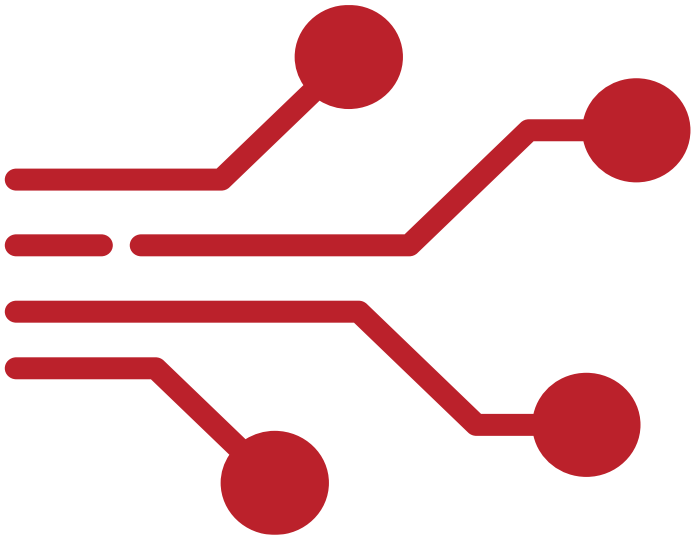
Jakość kodu



195

Dlaczego jakość kodu ma znaczenie?

- Jakość kodu wpływa na stabilność systemu
- Wysoka jakość kodu ułatwia rozwój oprogramowania
- Zły kod zwiększa o błędów i awarii
- Niska jakość kodu utrudnia utrzymanie systemu
- Błędy w kodzie mogą prowadzić do podatności bezpieczeństwa



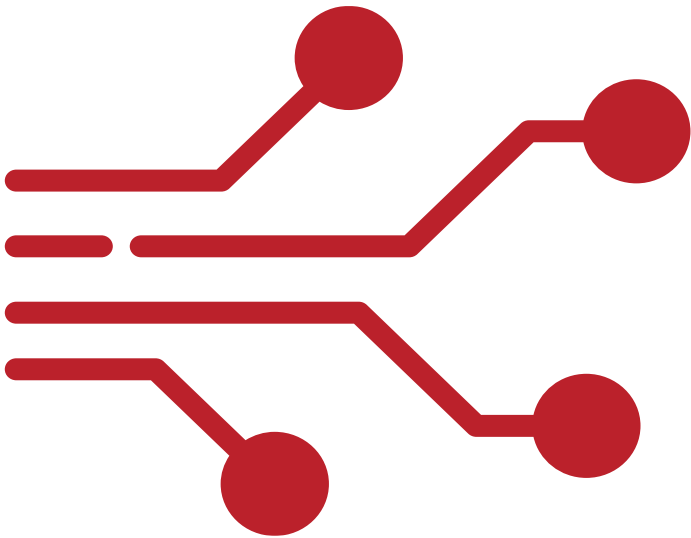
Konsekwencje złej jakości kodu

- Trudność w rozwoju i utrzymaniu systemu
- Wzrost liczby błędów w oprogramowaniu
- Wyższe koszty utrzymania systemu
- Problemy z wydajnością
- Powstawanie podatności bezpieczeństwa



Pojęcie jakości oprogramowania

- Jakość oprogramowania oznacza stopień spełnienia wymagań
- Dotyczy zarówno funkcjonalności, jak i właściwości technicznych systemu
- Obejmuje niezawodność, wydajność i bezpieczeństwo
- Zależy od jakości procesu wytwarzania oprogramowania
- Wpływa na możliwość utrzymania i rozwoju systemu



Jakość produktu a jakość kodu

- Jakość kodu jest podstawą jakości oprogramowania
- Błędy w kodzie wpływają na działanie systemu
- Dobry projekt nie wystarczy bez dobrej implementacji
- Kod musi być czytelny i zrozumiały dla zespołu
- Jakość kodu wpływa na bezpieczeństwo systemu



Czym jest dobry kod?

- Kod powinien być czytelny i zrozumiały
- Kod powinien być samokomentujący się
- Powinien być prosty i przejrzysty
- Powinien być łatwy do modyfikacji
- Powinien być możliwy do przetestowania
- Powinien minimalizować o błędów



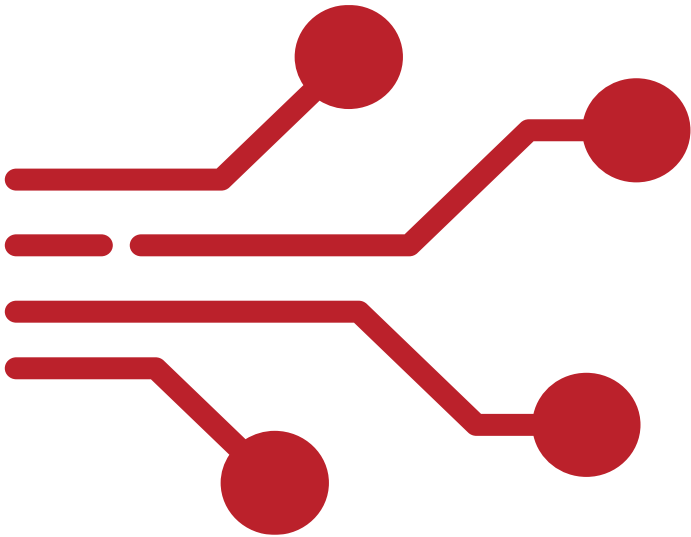
Czytelność kodu

- Kod powinien być łatwy do zrozumienia
- Nazwy zmiennych i funkcji powinny być jednoznaczne
- Struktura kodu powinna być przejrzysta
- Należy unikać nadmiernej złożoności
- Czytelność ułatwia utrzymanie systemu



Prostota kodu

- Kod powinien być możliwie prosty
- Należy unikać nadmiernie złożonych konstrukcji
- Jedna funkcja powinna realizować jedno zadanie
- Proste rozwiązania są łatwiejsze w utrzymaniu
- Złożoność zwiększa o błędów



Możliwość utrzymania kodu

- Kod powinien być łatwy do modyfikacji
- Zmiany nie powinny powodować błędów w innych częściach systemu
- Kod powinien być dobrze udokumentowany
- Struktura systemu powinna być logiczna
- Utrzymanie stanowi dużą część kosztów systemu



Testowalność kodu

- Kod powinien być możliwy do łatwego testowania
- Funkcje powinny mieć jasno określone wejścia i wyjścia
- Małe i niezależne moduły są łatwiejsze do testowania
- Testy pomagają wykrywać błędy we wczesnym etapie
- Testowalność zwiększa niezawodność systemu



Clean Code

- Clean Code oznacza czytelny i dobrze zaprojektowany kod
- Kod powinien być zrozumiały dla innych programistów
- Kod powinien być prosty i przejrzysty
- Należy unikać nadmiernej złożoności
- Celem jest łatwość utrzymania i rozwoju systemu



Najważniejsze zasady Clean Code

- Stosuj znaczące nazwy zmiennych i funkcji
- Funkcje powinny być krótkie i realizować jedno zadanie
- Unikaj powielania kodu
- Utrzymuj prostą strukturę programu
- Komentarze powinny wyjaśniać intencję, a nie kod



Zasady SOLID

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle



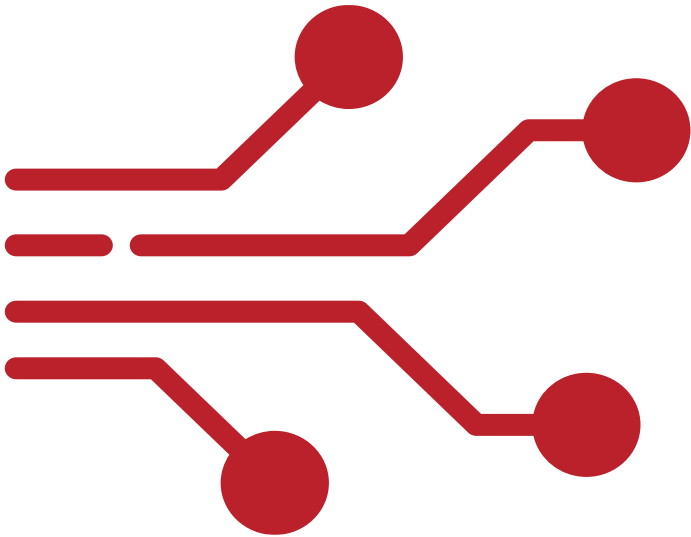
Zasady SOLID

- Single Responsibility Principle. Każda klasa powinna mieć jedną odpowiedzialność i realizować jedno zadanie. Zmiana w systemie powinna wymagać modyfikacji tylko jednej klasy.
- Open/Closed Principle. Kod powinien być otwarty na rozszerzanie, ale zamknięty na modyfikację. Nową funkcjonalność należy dodawać poprzez rozszerzanie systemu, a nie zmianę istniejącego kodu.
- Liskov Substitution Principle. Obiekty klas pochodnych powinny móc zastępować obiekty klas bazowych. Program powinien działać poprawnie niezależnie od tego, czy używana jest klasa bazowa czy jej rozszerzenie.

208

Zasady SOLID

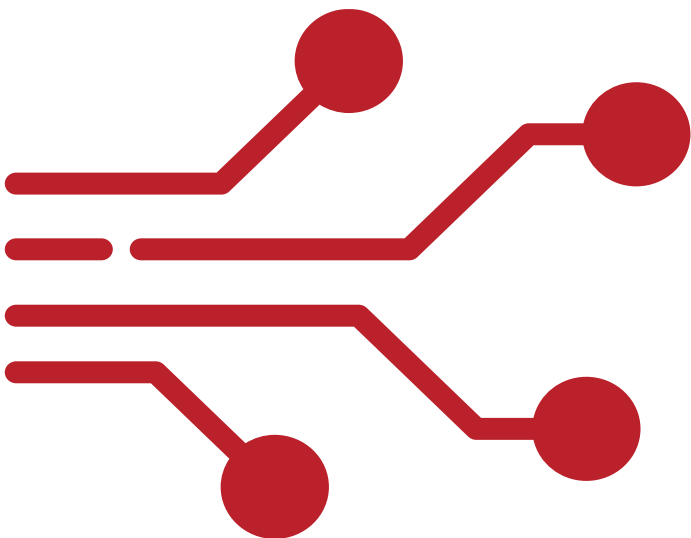
- Interface Segregation Principle. Lepiej tworzyć wiele małych, wyspecjalizowanych interfejsów, niż jeden duży. Klasy nie powinny być zmuszane do implementowania metod, których nie potrzebują.
- Dependency Inversion Principle. Moduły wysokiego poziomu nie powinny zależeć od modułów niskiego poziomu. Oba powinny zależeć od abstrakcji, co zmniejsza zależności w systemie.



Przykład złego i dobrego kodu

Przykład problematycznego kodu

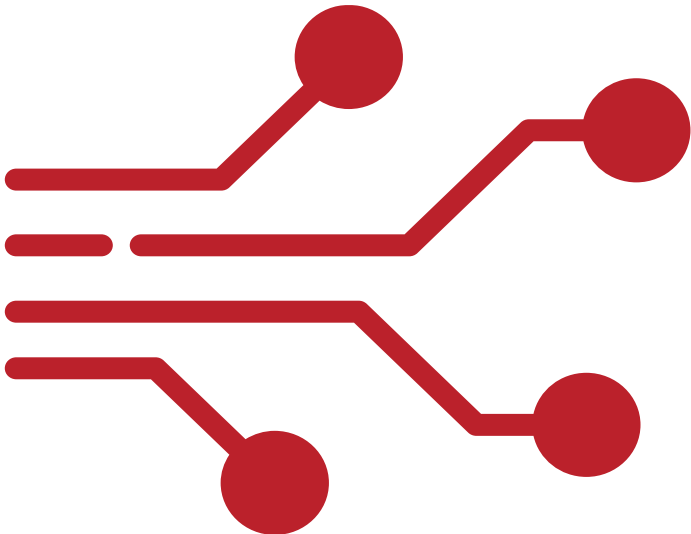
```
int d;  
if (a > b)  
{  
    d = a - b;  
}  
else  
{  
    d = b - a;  
}
```



210

Przykład złego i dobrego kodu

Lepsze rozwiązanie

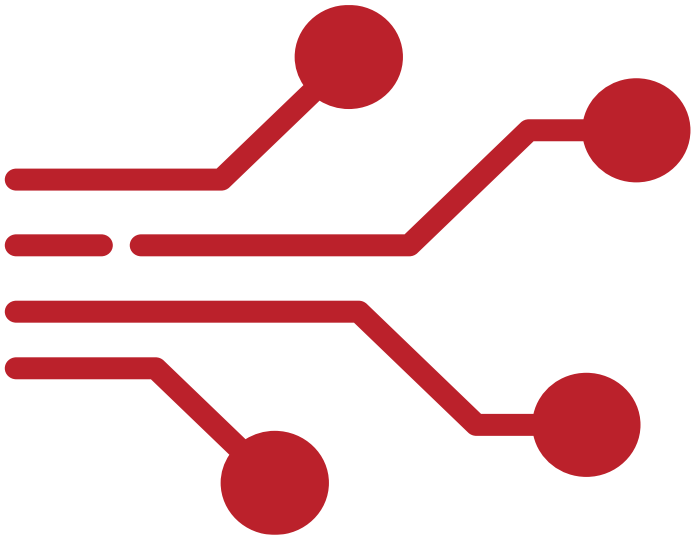


```
int difference = Math.Abs(a - b);
```

211

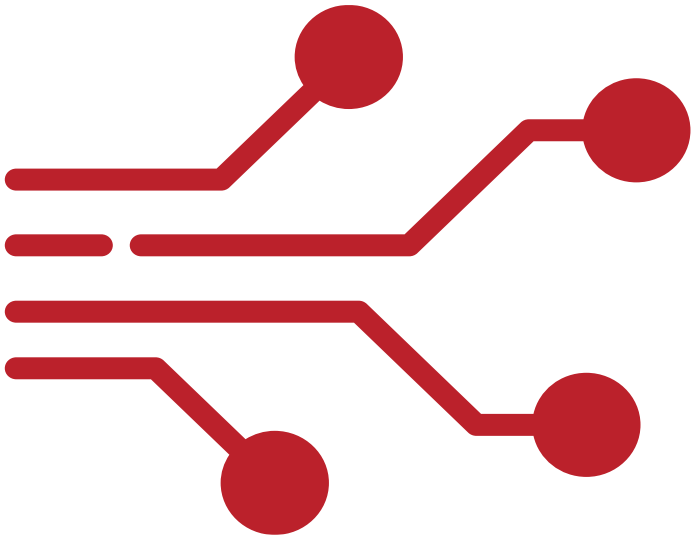
Typowe problemy w kodzie

- Zbyt długie funkcje
- Powielanie fragmentów kodu
- Zbyt duże klasy
- Złożona logika warunkowa
- Silne zależności między modułami



Code smells

- Code smells to symptomy problemów w kodzie
- Nie są bezpośrednio błędami programu
- Wskazują na potencjalne problemy projektowe
- Utrudniają utrzymanie i rozwój systemu
- Mogą prowadzić do powstawania błędów



Duże funkcje i klasy

- Funkcje o dużej liczbie linii kodu
- Klasy realizujące wiele różnych zadań
- Trudność w zrozumieniu działania kodu
- Problemy z testowaniem
- Naruszenie zasady pojedynczej odpowiedzialności



Powielanie kodu

- Ten sam fragment kodu występuje w wielu miejscach
- Zwiększa trudność utrzymania systemu
- Zmiany wymagają modyfikacji wielu fragmentów kodu
- Zwiększa o niespójności systemu
- Narusza zasadę DRY (Don't Repeat Yourself)



215

Zbyt silne zależności między klasami

- Klasy są silnie powiązane ze sobą
- Zmiana w jednej klasie wpływa na wiele innych
- Trudność w testowaniu modułów
- Ograniczona możliwość ponownego użycia kodu
- Utrudniona analiza bezpieczeństwa systemu



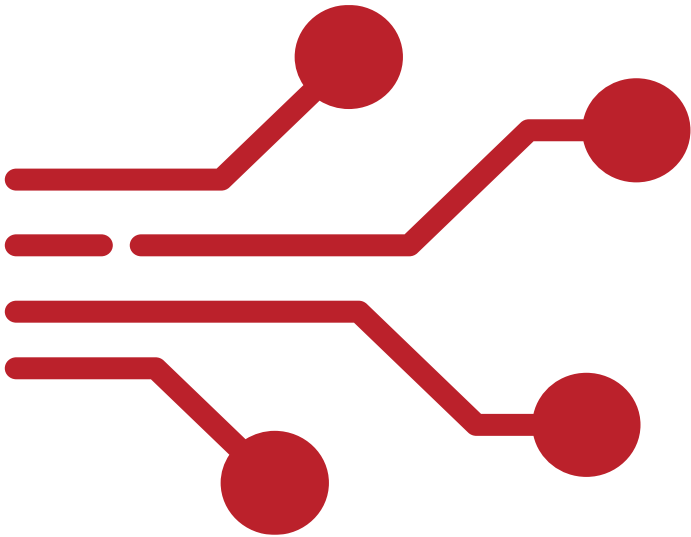
Określenia problemów w kodzie

- Hardcoded values. Wartości zapisane bezpośrednio w kodzie
- Utrudniają konfigurację i zwiększają ryzyko błędów
- Spaghetti code. Kod o bardzo złożonej i nieuporządkowanej strukturze. Trudny do zrozumienia i utrzymania
- Boilerplate code. Powtarzalny kod wymagany przez strukturę programu. Często nie wnosi logiki biznesowej
- Magic numbers. Liczby w kodzie bez wyjaśnienia ich znaczenia. Utrudniają zrozumienie działania programu

217

Najczęstsze błędy bezpieczeństwa w kodzie

- Hardcoded hasła i klucze API
- Brak walidacji danych wejściowych
- Brak obsługi wyjątków
- Pozostawiony kod debugujący
- Brak logowania zdarzeń bezpieczeństwa



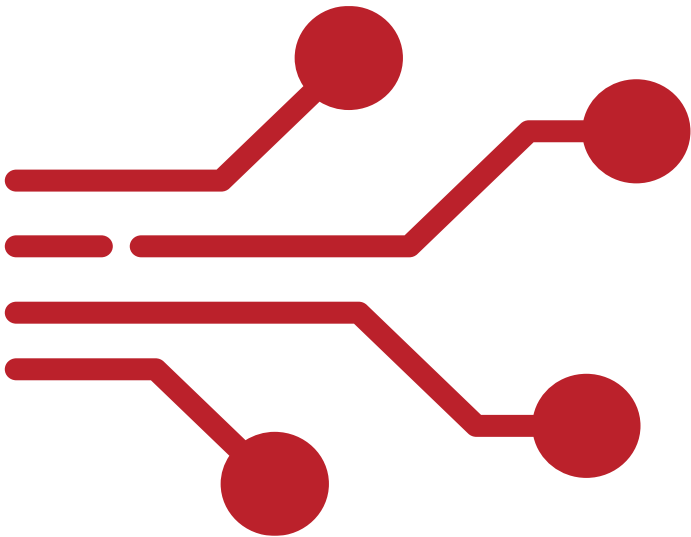
Bezpieczne programowanie

- Bezpieczne programowanie to tworzenie kodu odpornego na ataki
- Bezpieczeństwo powinno być uwzględnione już na etapie projektowania
- Programista musi zakładać, że dane wejściowe mogą być złośliwe
- Kod powinien minimalizować możliwość wykorzystania błędów
- Bezpieczeństwo jest elementem jakości oprogramowania

219

Najczęstsze podatności w kodzie

- SQL Injection
- Cross-Site Scripting (XSS)
- Niepoprawna walidacja danych wejściowych
- Niebezpieczna obsługa danych w pamięci
- Błędy kontroli dostępu



OWASP Top 10 a błędy programistyczne

- OWASP Top 10 to lista najczęstszych podatności aplikacji webowych
- Wiele z tych podatności wynika bezpośrednio z błędów w kodzie
- Najczęstsze problemy to brak walidacji danych i błędy uwierzytelniania
- Niewłaściwe zarządzanie sesją użytkownika
- Niebezpieczne przetwarzanie danych wejściowych

221

Przykład podatności: SQL Injection

- Niebezpieczny kod

```
string query = "SELECT * FROM users
```

```
WHERE login = '" + login +
```

```
"' AND password = '" + password + "'";
```

- Problem

- dane użytkownika są bezpośrednio wstawiane do zapytania SQL
- brak walidacji oraz brak zapytań parametryzowanych

- Możliwy atak

login: admin

password: ' OR '1'='1

Przykład podatności: Cross-Site Scripting (XSS)

- Niebezpieczny kod

```
string name = Request["name"];  
Response.Write("Hello " + name);
```

- Problem
 - dane użytkownika są wyświetlane bez filtrowania
 - możliwość wstrzyknięcia kodu JavaScript

- Możliwy atak

```
<script>  
alert("XSS attack");  
</script>
```

223

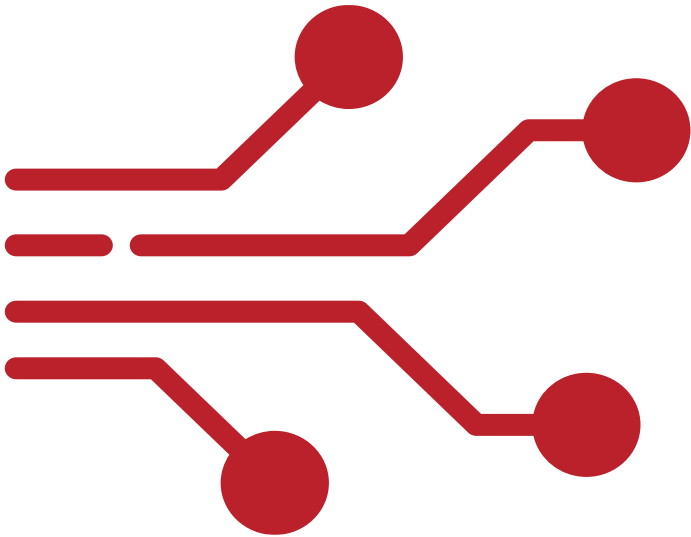
Secure Coding Practices

- Walidacja danych wejściowych
- Stosowanie zapytań parametryzowanych
- Bezpieczne zarządzanie hasłami
- Odpowiednia obsługa błędów
- Stosowanie aktualnych bibliotek i frameworków



Refaktoryzacja kodu

- Refaktoryzacja polega na poprawie struktury kodu
- Nie zmienia funkcjonalności programu
- Celem jest zwiększenie czytelności i jakości kodu
- Ułatwia dalszy rozwój systemu
- Pomaga usuwać code smells



225

Cele refaktoryzacji

- Zwiększenie czytelności kodu
- Uproszczenie struktury programu
- Usunięcie powielonego kodu
- Zmniejszenie zależności między modułami
- Ułatwienie testowania kodu



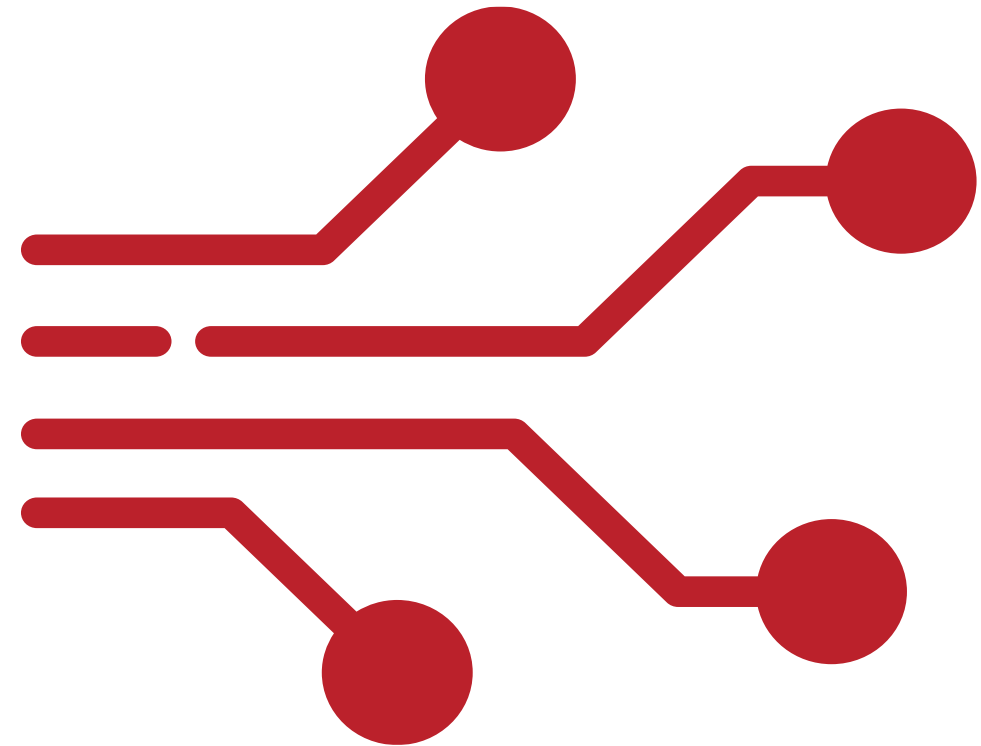
Przykłady refaktoryzacji

- Podział dużej funkcji na mniejsze funkcje
- Wyodrębnienie powtarzającego się kodu do osobnej metody
- Zastąpienie złożonych instrukcji warunkowych prostszą strukturą
- Wprowadzenie klas pomocniczych
- Poprawa nazw zmiennych i funkcji



227

Przeglądy kodu i dobre praktyki



228

Idea przeglądów kodu (Code Review)

- Przegląd kodu polega na analizie kodu przez innych programistów
- Celem jest wykrycie błędów oraz poprawa jakości kodu
- Code review jest częścią procesu wytwarzania oprogramowania
- Pozwala dzielić się wiedzą w zespole
- Może pomagać wykrywać podatności bezpieczeństwa

229

Cele przeglądów kodu

- Wykrywanie błędów w kodzie
- Poprawa czytelności i struktury kodu
- Weryfikacja zgodności ze standardami kodowania
- Wykrywanie potencjalnych podatności bezpieczeństwa
- Dzielenie się wiedzą w zespole



Proces przeglądu kodu

- Programista przygotowuje zmiany w kodzie
- Zmiany są zgłaszane do repozytorium
- Inni programiści analizują kod
- Zgłaszane są uwagi i sugestie
- Po poprawkach kod zostaje zaakceptowany



Rola recenzenta kodu

- Sprawdzenie poprawności działania kodu
- Ocena czytelności i struktury implementacji
- Weryfikacja zgodności ze standardami kodowania
- Identyfikacja potencjalnych błędów i podatności
- Proponowanie ulepszeń kodu



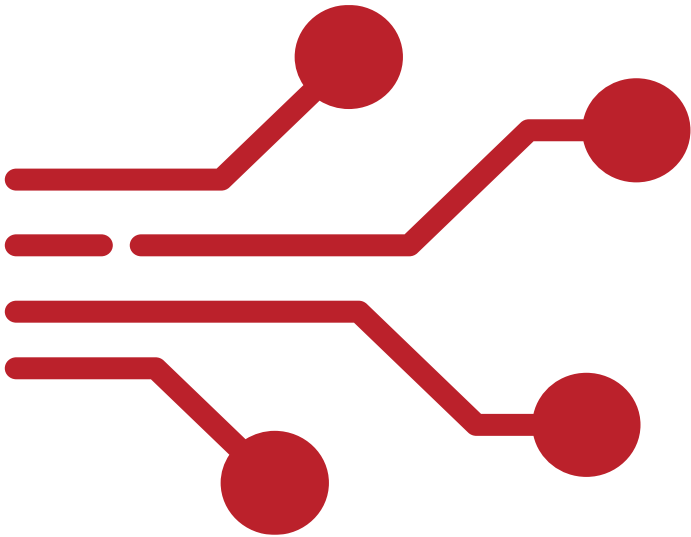
Najczęstsze problemy wykrywane w code review

- Błędy logiczne w implementacji
- Nieczytelne lub niejednoznaczne nazwy zmiennych
- Nadmiernie złożone fragmenty kodu
- Powielanie kodu
- Potencjalne podatności bezpieczeństwa



Statyczna analiza kodu

- Automatyczna analiza kodu bez jego uruchamiania
- Wykrywanie błędów i problemów w kodzie
- Analiza zgodności ze standardami kodowania
- Identyfikacja potencjalnych podatności bezpieczeństwa
- Wsparcie dla przeglądów kodu



SAST - analiza bezpieczeństwa kodu

- SAST oznacza Static Application Security Testing
- Automatyczna analiza bezpieczeństwa kodu źródłowego
- Wykrywanie potencjalnych podatności w kodzie
- Analiza wykonywana bez uruchamiania aplikacji
- Wykorzystywana w procesie Continuous Integration



235

Narzędzia analizy jakości i bezpieczeństwa kodu

- SonarQube
- Fortify
- Checkmarx
- ESLint/PyLint
- CodeQL



236

Praca zespołowa nad kodem

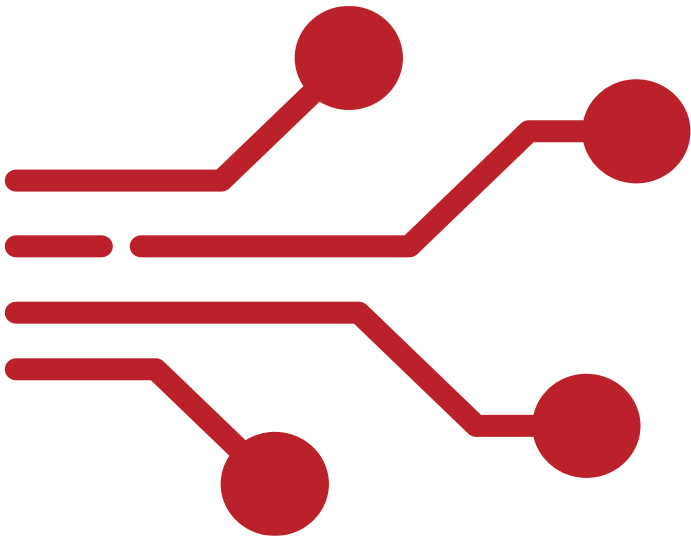
- Wspólne standardy kodowania
- Regularne przeglądy kodu
- Dzielenie się wiedzą w zespole
- Wspólna odpowiedzialność za jakość kodu
- Dokumentowanie ważnych decyzji projektowych



237

Jakość kodu w cyklu życia oprogramowania

- Jakość kodu wpływa na wszystkie etapy rozwoju systemu
- Wysoka jakość kodu ułatwia testowanie
- Ułatwia utrzymanie i rozwój oprogramowania
- Zmniejsza o błędów i podatności
- Wspiera długoterminowy rozwój systemu



Inspekcje Fagana

- Formalna metoda przeglądu artefaktów oprogramowania
- Opracowana w firmie IBM w latach 70.
- Służy do wykrywania błędów przed uruchomieniem programu
- Może być stosowana do kodu, dokumentacji i wymagań
- Opiera się na pracy zespołowej i formalnej procedurze przeglądu

239

Etapy inspekcji Fagana

- Planowanie inspekcji
- Przygotowanie uczestników
- Spotkanie inspekcyjne
- Rejestracja wykrytych defektów
- Poprawa błędów i weryfikacja

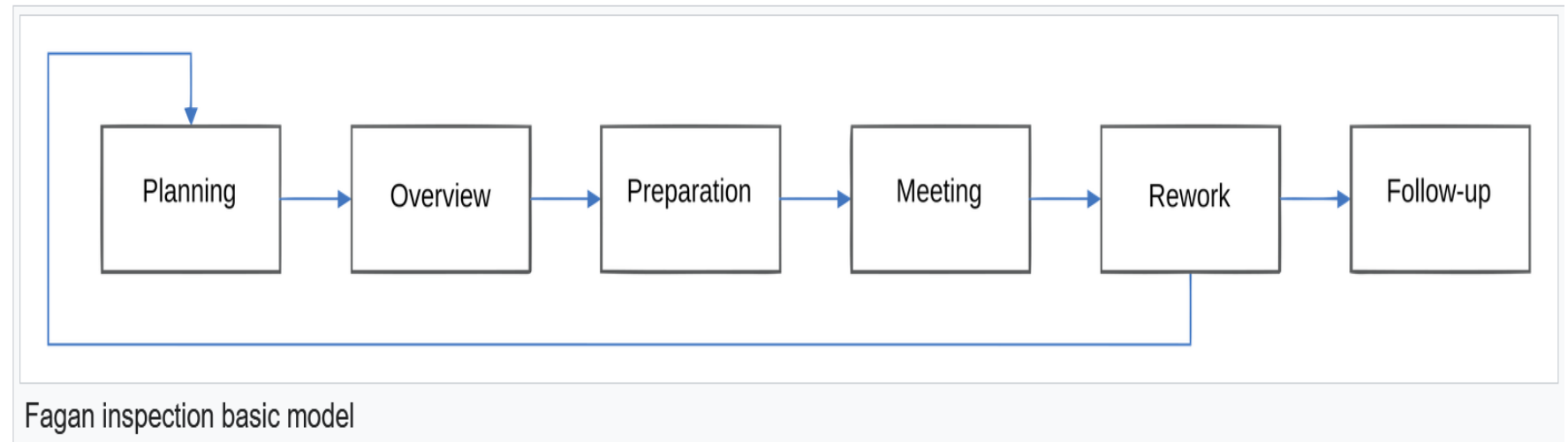


Role w inspekcji Fagana

- Moderator prowadzi proces inspekcji
- Autor - twórca analizowanego artefaktu
- Czytelnik prezentuje analizowany fragment
- Recenzenci identyfikują potencjalne błędy
- Sekretarz dokumentuje wykryte problemy



Model inspekcji Fagana



Rys. 10. Źródło: Wikipedia

242

Szacowanie liczby błędów - metoda dwukrotnego łowienia

- Metoda inspirowana statystyką populacji
- Stosowana do szacowania liczby błędów w systemie
- Polega na przeprowadzeniu dwóch niezależnych testów/inspekcji
- Analizuje się liczbę błędów wykrytych w obu testach/inspekcjach

243

Szacowanie liczby błędów - wzór

Niech

- N oznacza szacowaną liczbę błędów w systemie
- n1 będzie liczbą błędów wykrytych w pierwszym teście
- n2 liczbą błędów wykrytych w drugim teście
- Wtedy m to liczba błędów wykrytych w obu testach/inspekcjach

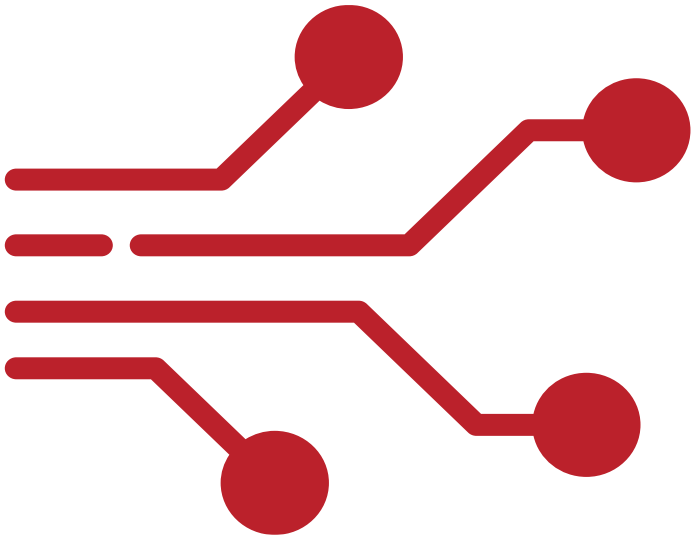
Szacowanie:

$$N = (n1 \times n2) / m$$

(w przybliżeniu)

Podsumowanie wykładu

- Jakość kodu ma kluczowe znaczenie dla jakości systemu
- Czytelny i prosty kod ułatwia rozwój oprogramowania
- Zasady Clean Code i SOLID pomagają tworzyć lepszy kod
- Przeglądy kodu poprawiają jakość i bezpieczeństwo systemu
- Narzędzia analizy kodu wspierają proces tworzenia oprogramowania



Dziękuję za uwagę

Wykład nr 6

Temat: Testowanie aplikacji

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

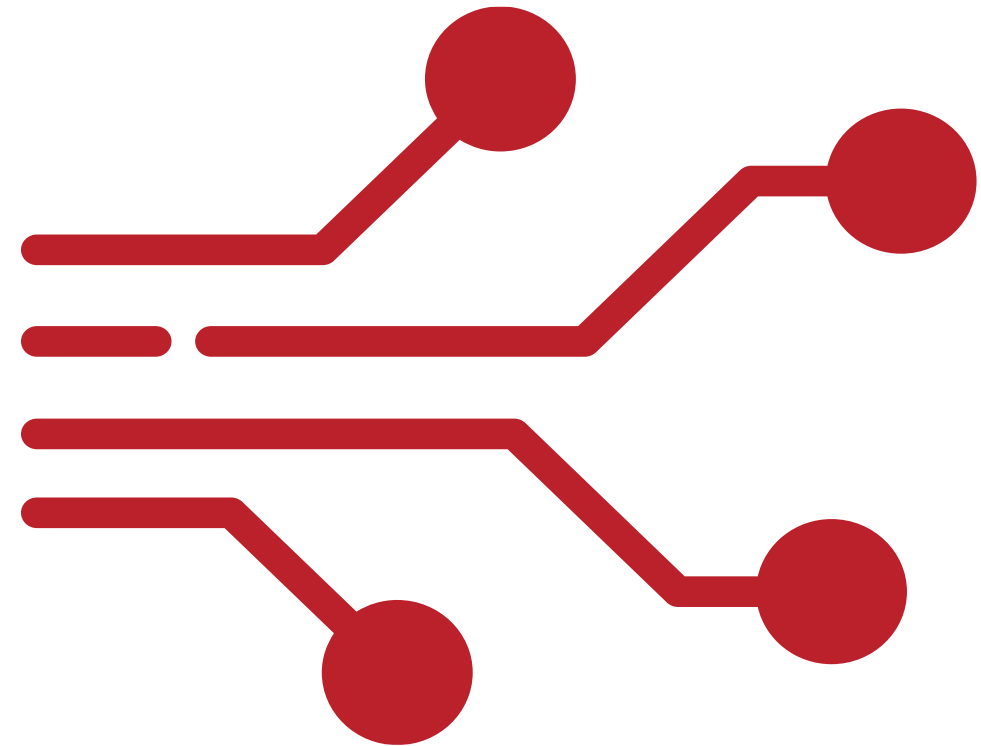
Plan prezentacji

- Rodzaje testów
- Automatyzacja
- xUnit



248

Rodzaje testów



249

Dlaczego testujemy oprogramowanie?

- Oprogramowanie może zawierać błędy
- Testowanie pozwala wykrywać defekty przed wdrożeniem
- Zmniejsza o awarii systemu
- Poprawia jakość i niezawodność aplikacji
- Wspiera bezpieczeństwo systemów informatycznych

250

Cele testowania

- Wykrywanie błędów w oprogramowaniu
- Weryfikacja poprawności działania systemu
- Sprawdzenie zgodności z wymaganiami
- Ocena jakości i niezawodności systemu
- Ograniczenie a awarii i podatności



251

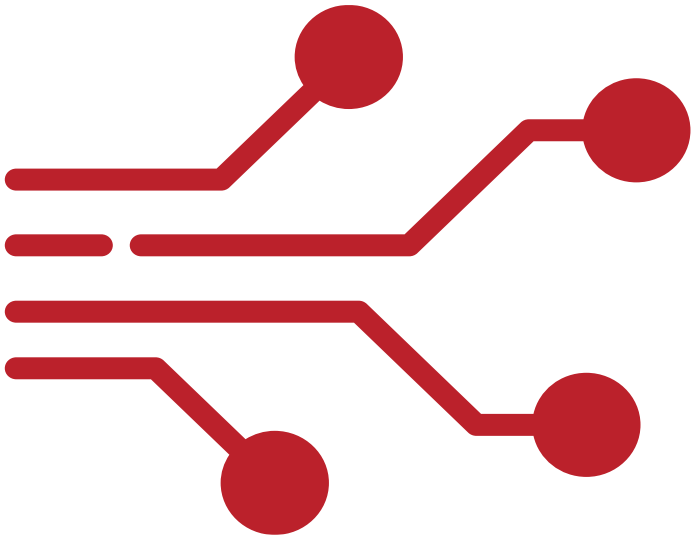
Testowanie a jakość oprogramowania

- Testowanie jest elementem zapewniania jakości oprogramowania
- Pomaga identyfikować błędy i defekty
- Umożliwia ocenę działania systemu w różnych scenariuszach
- Wspiera proces utrzymania i rozwoju systemu
- Zwiększa zaufanie do działania aplikacji



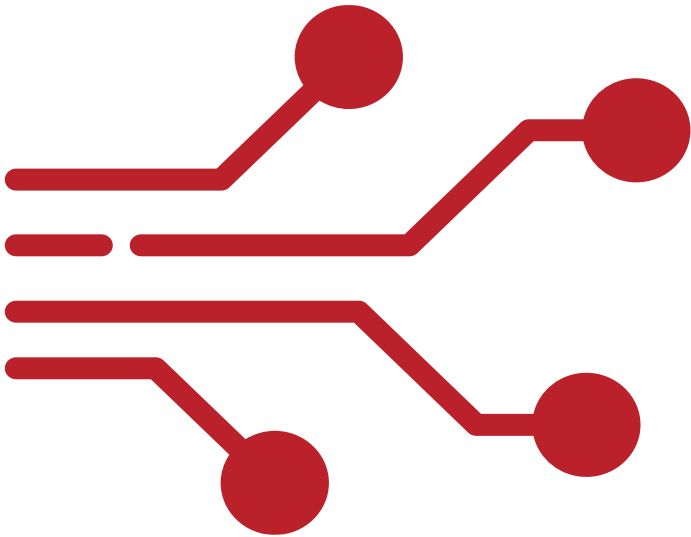
Koszt błędów w oprogramowaniu

- Koszt naprawy błędów rośnie wraz z etapem projektu
- Najtańsze jest wykrycie błędu na etapie projektowania
- Błędy wykryte po wdrożeniu mogą być bardzo kosztowne
- Awaria systemu może powodować straty finansowe
- W systemach krytycznych błędy mogą prowadzić do poważnych konsekwencji



Błąd, defekt i awaria

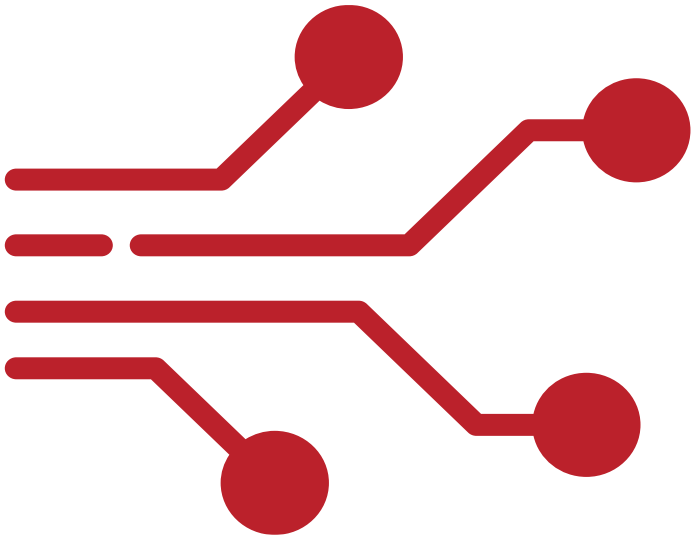
- Błąd (error) - pomyłka programisty w kodzie lub projekcie
- Defekt - niepoprawny fragment oprogramowania wynikający z błędu
- Awaria (failure) - nieprawidłowe działanie systemu podczas jego używania
- Jeden błąd może prowadzić do wielu defektów
- Defekt może, ale nie musi, powodować awarię



254

Scenariusz testowy

- Opisuje sposób sprawdzenia określonej funkcjonalności
- Przedstawia przebieg testu krok po kroku
- Określa warunki początkowe testu
- Definiuje oczekiwany rezultat działania systemu
- Może obejmować wiele przypadków testowych



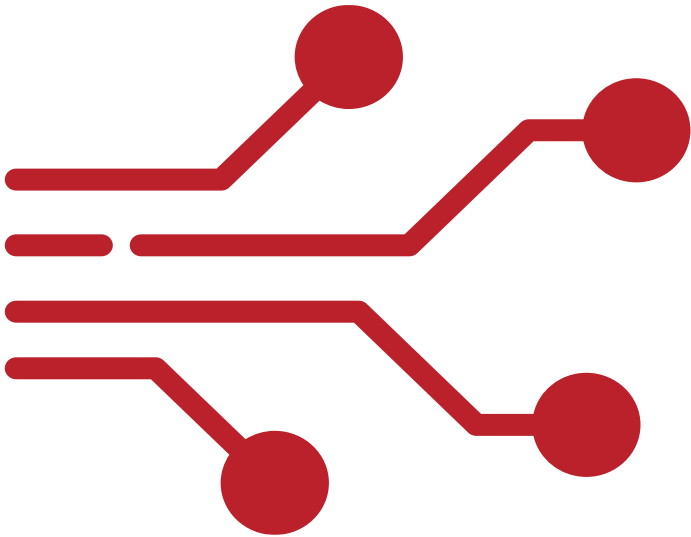
Przypadek testowy

- Szczegółowy opis pojedynczego testu
- Określa dane wejściowe testu
- Zawiera kroki wykonywane podczas testowania
- Definiuje oczekiwany wynik działania systemu
- Pozwala jednoznacznie ocenić wynik testu



Dane testowe

- Dane wykorzystywane podczas wykonywania testów
- Mogą być poprawne lub niepoprawne
- Powinny odzwierciedlać rzeczywiste dane użytkowników
- Umożliwiają sprawdzenie różnych scenariuszy działania systemu
- Mogą obejmować dane skrajne lub nietypowe



257

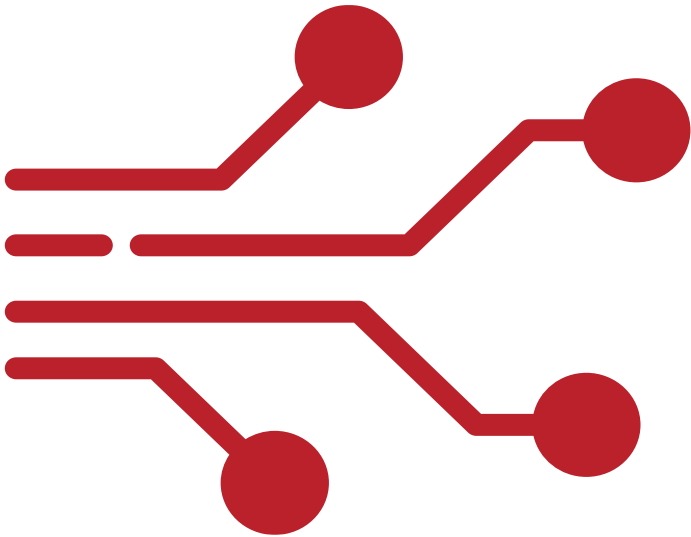
Testowanie jednostkowe

- Sprawdza działanie pojedynczych fragmentów kodu
- Dotyczy pojedynczych funkcji lub klas
- Wykonywane przez programistów
- Umożliwia szybkie wykrywanie błędów w kodzie
- Często jest automatyzowane



Testowanie integracyjne

- Sprawdza współpracę między modułami systemu
- Testuje komunikację między komponentami
- Wykrywa problemy integracyjne
- Może obejmować integrację z bazą danych lub API
- Wykonywane po testach jednostkowych



Testowanie systemowe

- Sprawdza działanie całego systemu
- Testuje wszystkie funkcjonalności aplikacji
- Wykonywane w środowisku zbliżonym do produkcyjnego
- Obejmuje różne scenariusze użycia systemu
- Weryfikuje zgodność z wymaganiami



Testowanie akceptacyjne

- Ostatni etap testowania systemu
- Sprawdza, czy system spełnia wymagania użytkowników
- Wykonywane przez użytkowników lub klienta
- Odbywa się przed wdrożeniem systemu
- Decyduje o akceptacji systemu



261

Testy funkcjonalne

- Sprawdzają poprawność działania funkcji systemu
- Weryfikują realizację wymagań funkcjonalnych
- Testują reakcję systemu na określone dane wejściowe
- Sprawdzają poprawność wyników działania programu
- Koncentrują się na funkcjonalności aplikacji



Testy niefunkcjonalne

- Sprawdzają jakość działania systemu
- Dotyczą wydajności, niezawodności i bezpieczeństwa
- Analizują zachowanie systemu w różnych warunkach
- Obejmują testy obciążeniowe i wydajnościowe
- Sprawdzają odporność systemu na błędy



Testy wydajnościowe

- Sprawdzają szybkość działania systemu
- Analizują czas odpowiedzi aplikacji
- Testują działanie systemu przy dużym obciążeniu
- Pozwalają identyfikować wąskie gardła systemu
- Obejmują testy obciążeniowe i testy przeciążeniowe



Testy bezpieczeństwa

- Sprawdzają odporność systemu na ataki
- Identyfikują podatności bezpieczeństwa
- Weryfikują mechanizmy uwierzytelniania i autoryzacji
- Analizują ochronę danych w systemie
- Obejmują testy penetracyjne i testy podatności



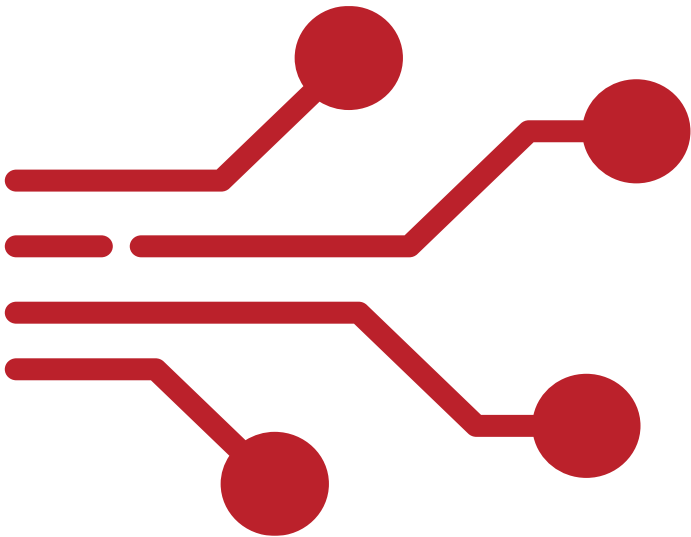
Testowanie czarnej skrzynki

- Tester nie zna wewnętrznej struktury systemu
- Testowanie opiera się na danych wejściowych i wynikach
- Sprawdza zgodność działania systemu z wymaganiami
- Nie wymaga znajomości kodu źródłowego
- Koncentruje się na zachowaniu systemu



Testowanie białej skrzynki

- Tester zna strukturę wewnętrzną systemu
- Analizowana jest logika programu
- Sprawdzane są ścieżki wykonania kodu
- Wykorzystywana jest znajomość kodu źródłowego
- Często stosowane w testach jednostkowych



267

Dobór danych testowych - wartości graniczne

- Błędy często pojawiają się na granicach zakresów danych
- Testy powinny obejmować wartości skrajne
- Sprawdzane są wartości minimalne i maksymalne
- Analizowane są wartości tuż przed i tuż po granicy

Przykład:

Zakres danych: 1 - 100

Testowane wartości: 0, 1, 2, 99, 100, 101

Partycjonowanie równoważności

- Technika projektowania przypadków testowych
- Polega na podziale danych wejściowych na klasy równoważności
- Zakłada, że elementy tej samej klasy zachowują się podobnie
- Testowanie jednej wartości reprezentuje całą klasę
- Pozwala ograniczyć liczbę testów



Przykład partycjonowania równoważności

Założmy: wiek użytkownika: 0-120

Klasy równoważności:

- wartości poprawne: 0-120
- wartości poniżej zakresu: < 0
- wartości powyżej zakresu: > 120

Przykładowe dane testowe:

-1

25

121

Partycjonowanie równoważności a wartości graniczne

- Partycjonowanie dzieli dane na klasy
- Analiza wartości granicznych bada granice klas
- Obie techniki są często stosowane razem
- Pozwalają zmniejszyć liczbę testów przy zachowaniu skuteczności



271

Pokrycie kodu (code coverage)

- Metryka określająca zakres przetestowania kodu
- Pokazuje, jaka część programu została wykonana podczas testów
- Pomaga ocenić jakość zestawu testów
- Nie gwarantuje braku błędów
- Jest używana w analizie jakości oprogramowania



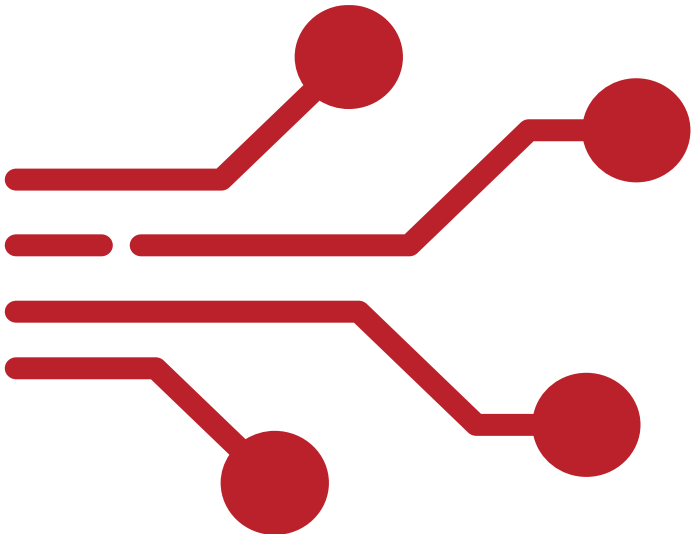
272

Pokrycie instrukcji

- Sprawdza, czy każda instrukcja programu została wykonana
- Najprostsza metryka pokrycia kodu
- Wyrażana jako procent wykonanych instrukcji

Wzór:

$\text{pokrycie} = \frac{\text{wykonane instrukcje}}{\text{wszystkie instrukcje}}$



Pokrycie gałęzi

- Sprawdza, czy wykonano wszystkie możliwe gałęzie programu
- Dotyczy instrukcji warunkowych
- Analizuje zarówno ścieżkę prawdziwą, jak i fałszywą

Przykład:

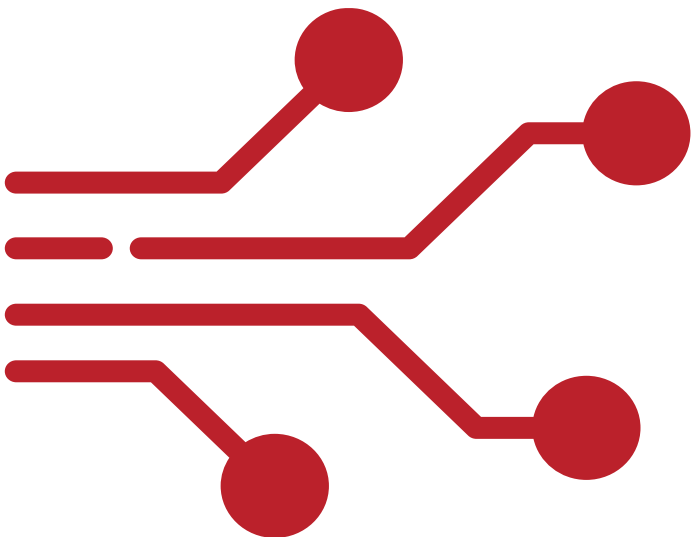
if ($x > 0$)

A

else

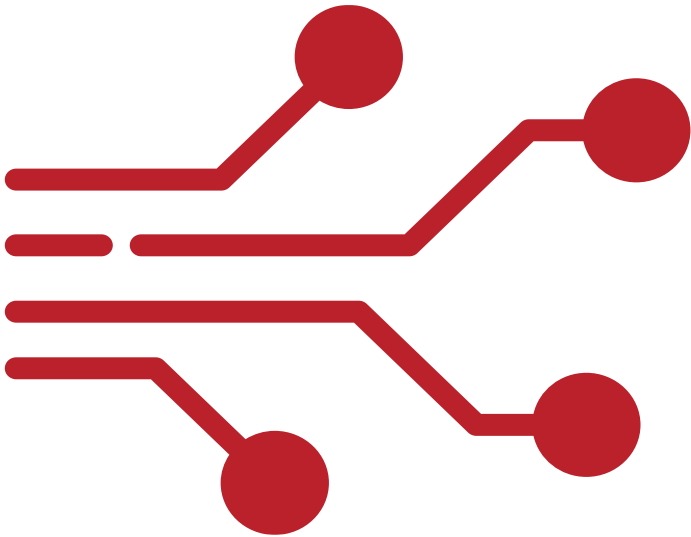
B

Testy powinny wykonać zarówno A, jak i B.



Pokrycie ścieżek

- Sprawdza wszystkie możliwe ścieżki wykonania programu
- Jest bardzo wymagającą metryką
- Liczba ścieżek może rosnąć bardzo szybko
- W praktyce pełne pokrycie ścieżek jest rzadko osiągalne



275

Przykładowe narzędzia analizy pokrycia kodu

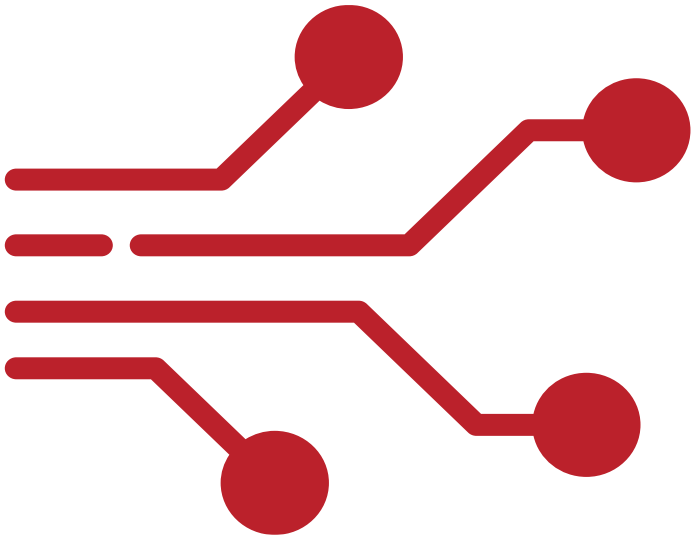
- Visual Studio Code Coverage
- Coverlet
- dotCover
- JaCoCo
- Istanbul



276

Czy 100% pokrycia kodu oznacza dobre testy?

- Wysokie pokrycie kodu nie gwarantuje poprawności programu
- Test może wykonać kod, ale nie sprawdzić wyniku
- Możliwe jest sztuczne zwiększanie pokrycia
- Pokrycie kodu nie wykrywa błędów logicznych
- Metryka powinna być używana z innymi technikami testowania
- Przykład: równanie kwadratowe

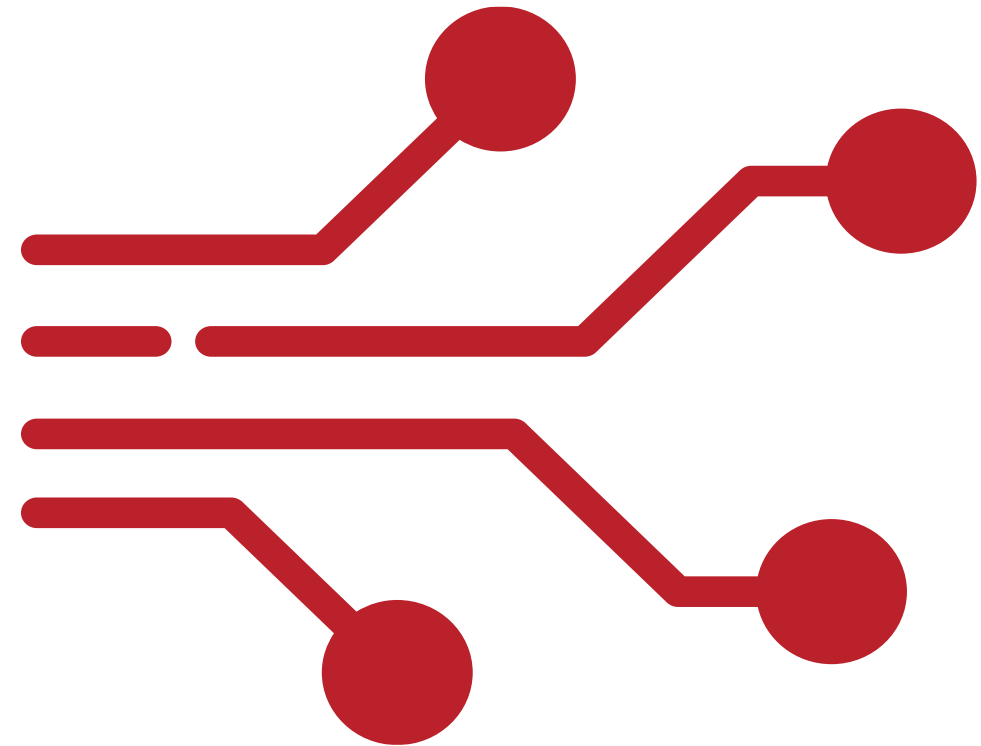


Testowanie szarej skrzynki

- Tester posiada częściową wiedzę o systemie
- Łączy elementy testowania czarnej i białej skrzynki
- Wykorzystuje wiedzę o architekturze systemu
- Pozwala tworzyć bardziej realistyczne scenariusze testów
- Często stosowane w testach bezpieczeństwa



Automatyzacja



279

Testy manualne i automatyczne

- Testy manualne - wykonywane są przez testerów
- Testy automatyczne - wykonywane są przez programy testujące
- Automatyzacja pozwala wielokrotnie uruchamiać testy
- Testy manualne są przydatne przy analizie interfejsu użytkownika
- Testy automatyczne przyspieszają proces testowania

280

Frameworki testowe

- Ułatwiają tworzenie i wykonywanie testów
- Automatyzują proces testowania
- Pozwalają definiować przypadki testowe w kodzie
- Integrują się z systemami kontroli wersji i CI
- Ułatwiają raportowanie wyników testów



281

Continuous Testing

- Automatyczne testowanie podczas rozwoju oprogramowania
- Testy uruchamiane przy każdej zmianie w kodzie
- Integracja z procesem Continuous Integration
- Szybkie wykrywanie błędów i regresji
- Wspiera utrzymanie wysokiej jakości systemu



282

Testy bezpieczeństwa aplikacji

- Sprawdzają odporność aplikacji na ataki
- Identyfikują potencjalne podatności
- Analizują mechanizmy uwierzytelniania i autoryzacji
- Sprawdzają ochronę danych użytkowników
- Oceniają bezpieczeństwo komunikacji systemu



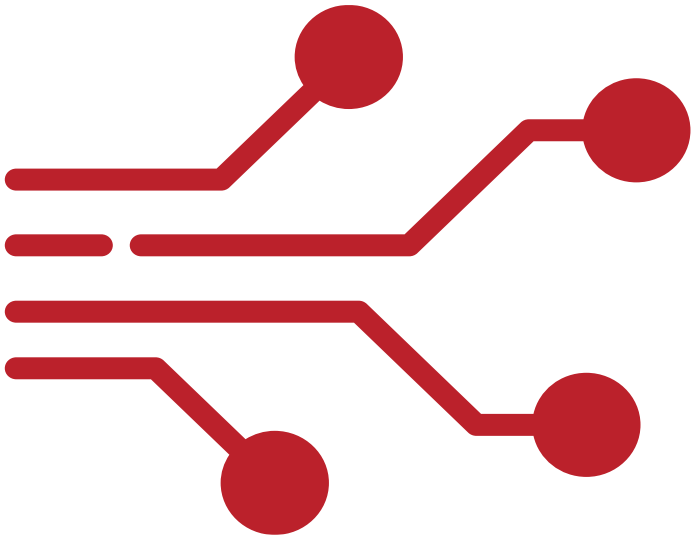
Fuzz testing

- Polega na wysyłaniu losowych lub niepoprawnych danych do systemu
- Sprawdza odporność aplikacji na nietypowe dane wejściowe
- Pozwala wykrywać błędy w obsłudze danych
- Może prowadzić do wykrycia podatności bezpieczeństwa
- Często stosowany w testowaniu systemów krytycznych

284

Testy penetracyjne

- Symulują rzeczywiste ataki na system
- Wykonywane przez specjalistów bezpieczeństwa
- Identyfikują podatności w aplikacjach i infrastrukturze
- Sprawdzają skuteczność mechanizmów ochrony
- Pozwalają ocenić poziom bezpieczeństwa systemu



Testowanie podatności

- Polega na identyfikowaniu znanych podatności w systemie
- Wykorzystuje bazy znanych podatności bezpieczeństwa
- Może być wykonywane automatycznie
- Analizuje konfigurację systemu i aplikacji
- Pomaga wykrywać słabe punkty systemu



Narzędzia do testów jednostkowych

- JUnit – dedykowany dla języka Java. Pozwala definiować metody testowe, sprawdzać oczekiwane wyniki oraz automatyzować weryfikację działania klas i metod
- NUnit - framework do testów jednostkowych dla platformy .NET
- PyTest – narzędzie dla Pythona
- TestNG - stanowi bardziej rozbudowaną alternatywę dla JUnit w niektórych zastosowaniach
- xUnit – odpowiedniki Junit dla innych języków

287

Selenium

- Narzędzie do automatyzacji testów aplikacji webowych
- Umożliwia symulowanie działań użytkownika w przeglądarce
- Obsługuje wiele języków programowania
- Może być integrowane z frameworkami testowymi
- Wspiera automatyczne testy interfejsu użytkownika

288

OWASP ZAP

- Narzędzie do testowania bezpieczeństwa aplikacji webowych
- Automatycznie wykrywa podatności w aplikacjach
- Analizuje ruch HTTP między przeglądarką a serwerem
- Wspiera testy penetracyjne aplikacji webowych
- Jest rozwijane przez społeczność OWASP



Burp Suite

- Narzędzie do testów bezpieczeństwa aplikacji webowych
- Analizuje i modyfikuje ruch HTTP i HTTPS
- Umożliwia wykrywanie podatności bezpieczeństwa
- Wspiera testy penetracyjne aplikacji
- Często wykorzystywane przez specjalistów bezpieczeństwa



Planowanie testów

- Określenie zakresu testowania
- Identyfikacja funkcjonalności do przetestowania
- Wybór metod i narzędzi testowania
- Określenie harmonogramu testów
- Przydzielenie odpowiedzialności w zespole



Projektowanie testów

- Tworzenie scenariuszy testowych
- Przygotowanie przypadków testowych
- Określenie danych testowych
- Definiowanie oczekiwanych wyników
- Przygotowanie środowiska testowego



Wykonywanie testów

- Realizacja przygotowanych przypadków testowych
- Wprowadzanie danych testowych do systemu
- Obserwacja zachowania aplikacji
- Porównanie wyników z oczekiwanymi rezultatami
- Rejestrowanie wykrytych błędów



Raportowanie wyników testów

- Dokumentowanie wyników testów
- Rejestrowanie wykrytych błędów
- Analiza przyczyn problemów
- Priorytetyzacja wykrytych defektów
- Przekazywanie informacji zespołowi projektowemu



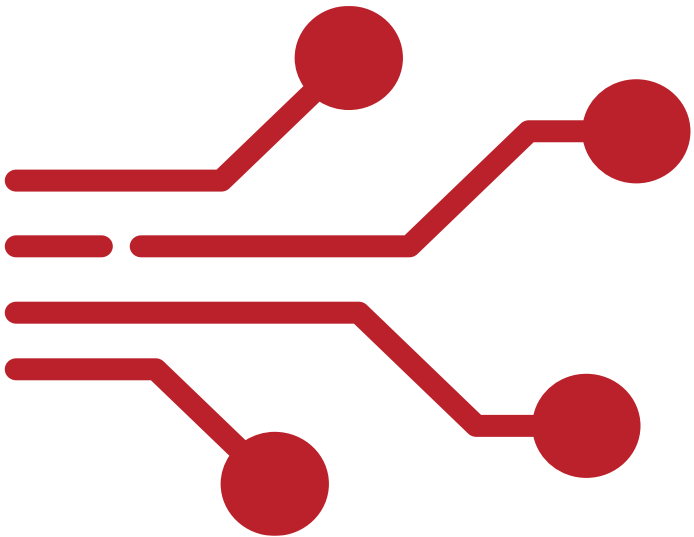
Najważniejsze zasady testowania

- Testowanie wykazuje obecność błędów, ale nie ich brak
- Nie jest możliwe przetestowanie wszystkich przypadków
- Testowanie powinno rozpoczynać się jak najwcześniej
- Testy powinny być systematyczne i zaplanowane
- Należy koncentrować się na obszarach największego ryzyka



Testowanie w cyklu życia oprogramowania

- Testowanie jest częścią całego procesu wytwarzania oprogramowania
- Testy są wykonywane na różnych etapach projektu
- Każdy etap rozwoju systemu wymaga odpowiednich testów
- Testowanie wspiera utrzymanie i rozwój systemu
- Pozwala ograniczać ryzyko błędów w kolejnych wersjach



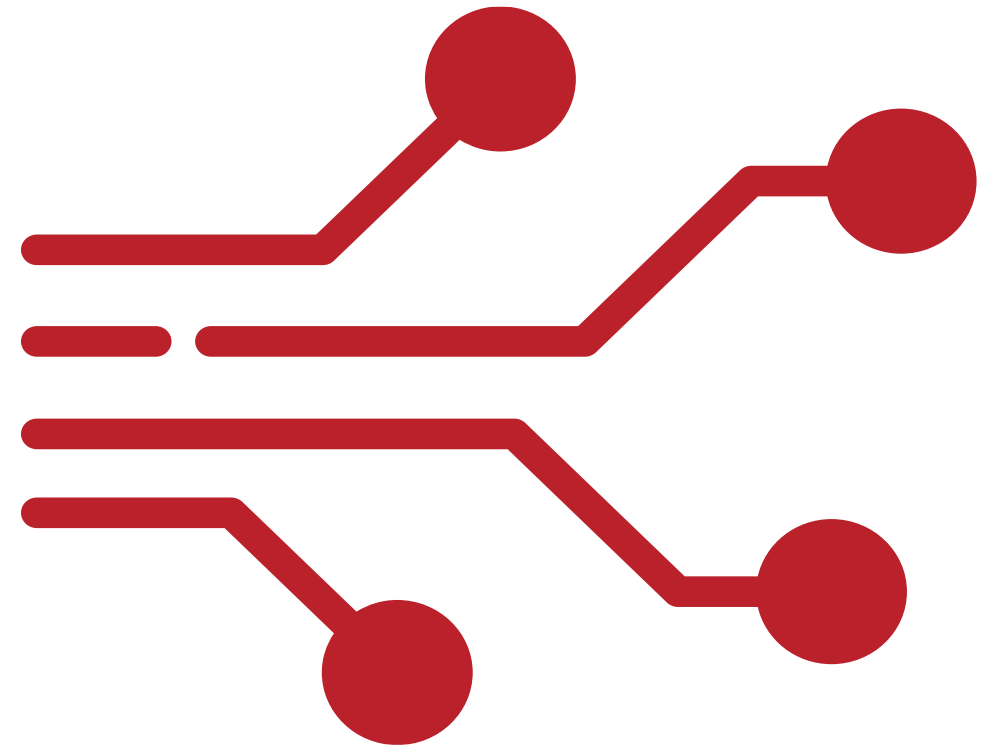
296

Testowanie a bezpieczeństwo systemu

- Błędy w oprogramowaniu mogą prowadzić do podatności
- Testowanie pomaga identyfikować słabe punkty systemu
- Testy bezpieczeństwa uzupełniają klasyczne testowanie
- Automatyczne narzędzia wspierają analizę bezpieczeństwa
- Bezpieczeństwo powinno być uwzględnione w całym procesie testowania

297

xUnit



298

xUnit w C#

- xUnit jest frameworkiem do testów automatycznych w .NET
- Służy głównie do testów jednostkowych
- Testy zapisuje się jako metody w klasach testowych
- Framework uruchamia testy i raportuje wyniki
- Często integrowany jest z Visual Studio i CI/CD



Jak wygląda test w xUnit?

- Test znajduje się w osobnej metodzie
- Metoda testowa oznaczana jest atrybutem [Fact]
- Dla wielu zestawów danych można używać [Theory]
- Wynik testu sprawdzany jest przy pomocy asercji
- Popularna konwencja: Arrange - Act - Assert

300

Przykład prostego testu xUnit

using Xunit;

public class CalculatorTests

{

[Fact]

public void Add_TwoNumbers_ReturnsCorrectSum()

{

// Arrange

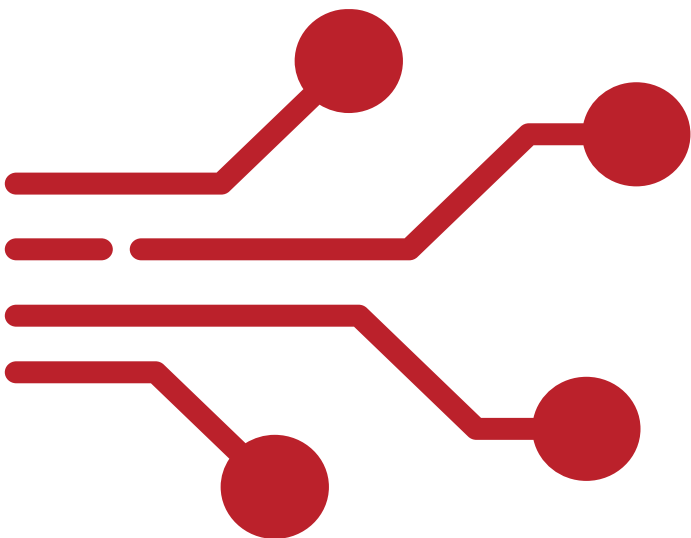
var calculator = new Calculator();

// Act

int result = calculator.Add(2, 3);

// Assert

Assert.Equal(5, result);}}



301

Przykład prostego testu xUnit

- [Fact] oznacza pojedynczy test
- Assert.Equal sprawdza oczekiwany wynik
- Nazwa testu powinna opisywać jego cel
- Test powinien być krótki i jednoznaczny



Testy z danymi wejściowymi - [Theory]

using Xunit;

public class CalculatorTests

{

[Theory]

[InlineData(2, 3, 5)]

[InlineData(10, 5, 15)]

[InlineData(-1, 1, 0)]

public void Add_VariousNumbers_ReturnsCorrectSum(int a, int b, int expected)

{

var calculator = new Calculator();

int result = calculator.Add(a, b);

Assert.Equal(expected, result);}}

Testy z danymi wejściowymi - [Theory]

- [Theory] pozwala uruchomić test wiele razy
- [InlineData] przekazuje dane wejściowe
- Jeden test może sprawdzić wiele scenariuszy
- Zmniejsza powielanie kodu testowego



Dobre praktyki testowania w C#

- Jeden test powinien sprawdzać jedną rzecz
- Testy powinny być krótkie i czytelne
- Nazwy testów powinny być opisowe
- Należy unikać zależności między testami
- Testy powinny działać automatycznie i powtarzalnie



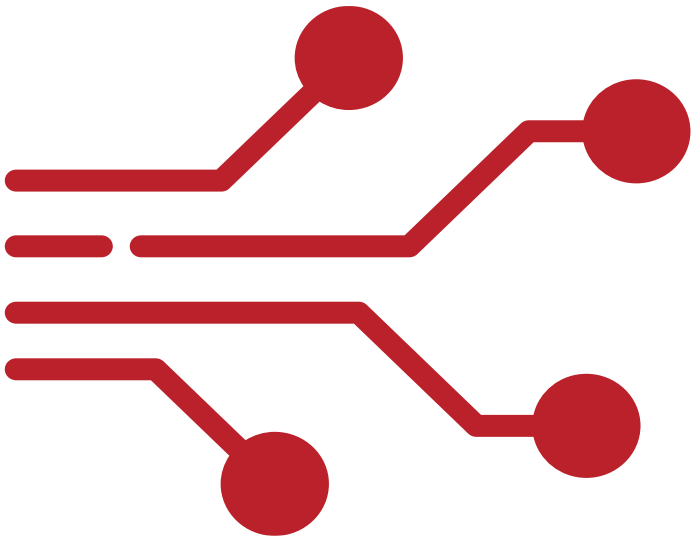
Mocki w testach jednostkowych

- Mock symuluje działanie zależności w teście
- Pozwala testować klasę w izolacji
- Zastępuje bazę danych, API lub inne usługi
- Umożliwia kontrolowanie zachowania zależności
- Popularne biblioteki: Moq, Nsubstitute



Pipeline testów w projekcie .NET

- Programista wprowadza zmianę w repozytorium
- System CI buduje aplikację
- Automatycznie uruchamiane są testy jednostkowe
- Uruchamiana jest analiza jakości i bezpieczeństwa kodu
- Po przejściu testów aplikacja może zostać wdrożona



307

Dlaczego testy jednostkowe często nie działają w projektach?

- Brak czasu na pisanie testów
- Zbyt silne zależności między klasami
- Kod trudny do testowania
- Brak kultury testowania w zespole
- Testy nie są utrzymywane wraz z kodem



Podsumowanie wykładu

- Testowanie jest kluczowym elementem jakości oprogramowania
- Istnieją różne poziomy testowania systemów
- Testy mogą być funkcjonalne i нефunkcjonalne
- Automatyzacja testów wspiera rozwój nowoczesnych systemów
- Testowanie bezpieczeństwa pomaga identyfikować podatności



Dziękuję za uwagę

Wykład nr 7

Temat: Tworzenie dokumentacji oprogramowania

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

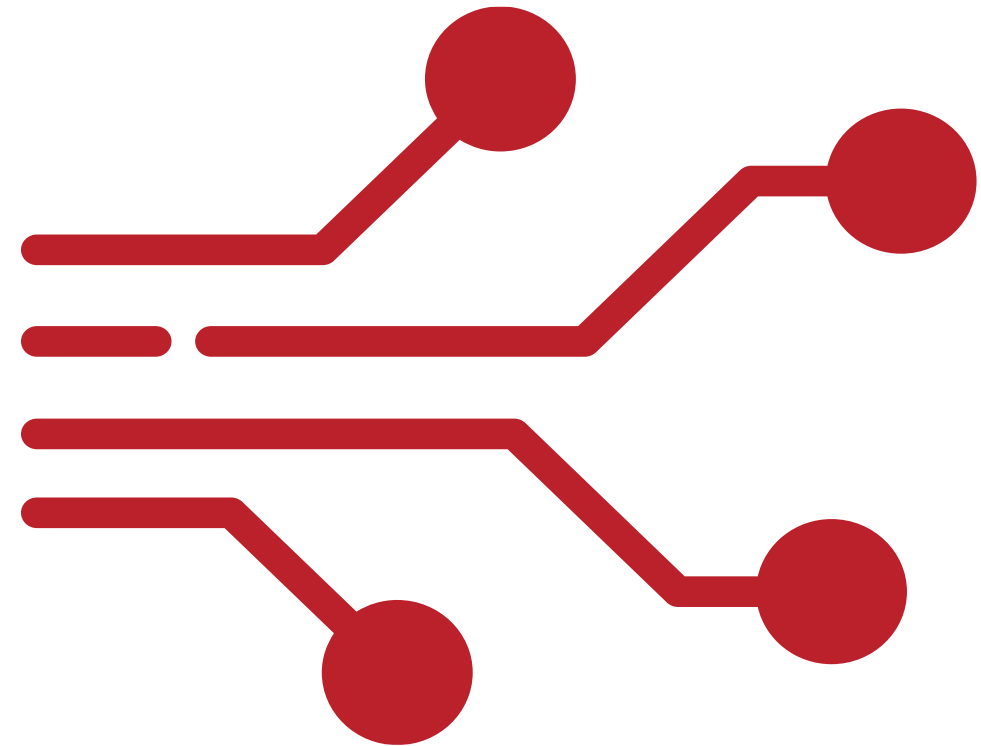
Plan prezentacji

- Wprowadzenie
- Rodzaje dokumentacji



312

Wprowadzenie



313

Dlaczego dokumentacja jest potrzebna?

- Ułatwia zrozumienie działania systemu
- Wspiera rozwój i utrzymanie oprogramowania
- Umożliwia współpracę w zespole programistycznym
- Pozwala szybciej wdrażać nowych programistów
- Pomaga analizować i rozwiązywać problemy



Rodzaje dokumentacji oprogramowania

- Dokumentacja techniczna systemu
- Dokumentacja użytkownika
- Dokumentacja administratora systemu
- Dokumentacja kodu źródłowego
- Dokumentacja architektury systemu



Dokumentacja a cykl życia oprogramowania

- Dokumentacja powstaje na różnych etapach projektu
- Wspiera analizę, projektowanie i implementację systemu
- Jest aktualizowana w trakcie rozwoju oprogramowania
- Pomaga w utrzymaniu systemu po wdrożeniu
- Ułatwia rozwój kolejnych wersji systemu



316

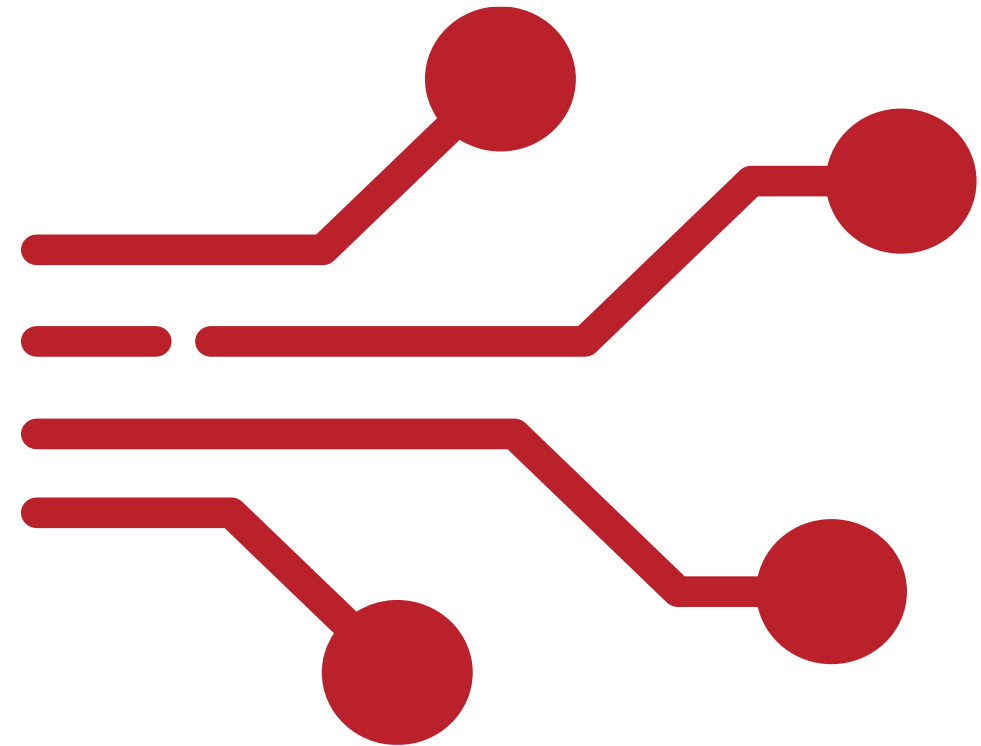
Koszt braku dokumentacji

- Trudności w zrozumieniu działania systemu
- Wolniejsze wprowadzanie zmian w kodzie
- Problemy z utrzymaniem systemu
- Trudności we wdrażaniu nowych programistów
- Większe o błędów i podatności



317

Rodzaje dokumentacji



318

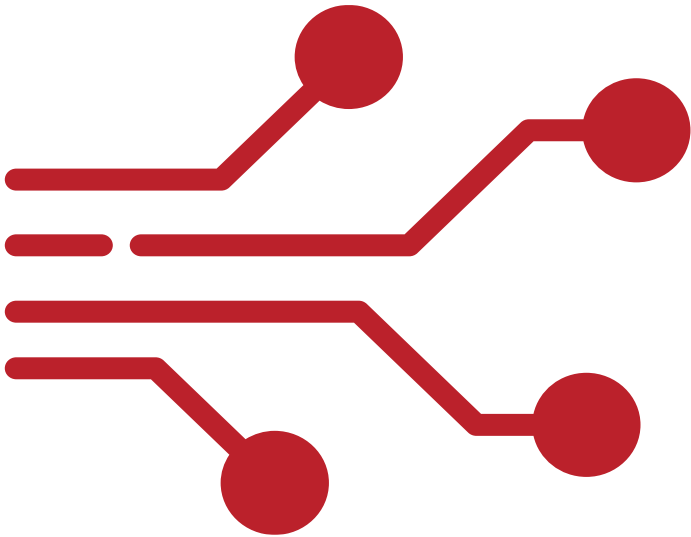
Dokumentacja architektury systemu

- Opisuje ogólną strukturę systemu
- Przedstawia główne komponenty aplikacji
- Pokazuje relacje między modułami
- Dokumentuje technologie wykorzystane w systemie
- Wspiera rozwój i utrzymanie oprogramowania



Diagramy architektury

- Przedstawiają strukturę systemu w formie graficznej
- Pokazują komponenty oraz ich zależności
- Ułatwiają zrozumienie działania systemu
- Wspierają analizę architektury oprogramowania
- Są ważnym elementem dokumentacji technicznej



320

Dokumentacja modułów systemu

- Opisuje funkcjonalność poszczególnych modułów
- Przedstawia odpowiedzialność każdego komponentu
- Dokumentuje interfejsy między modułami
- Ułatwia rozwój i utrzymanie systemu
- Pomaga analizować zależności w systemie



321

Dokumentacja interfejsów API

- Opisuje sposób komunikacji z systemem
- Dokumentuje dostępne operacje i endpointy
- Określa format danych wejściowych i wyjściowych
- Zawiera przykłady użycia API
- Ułatwia integrację z innymi systemami



Instrukcja użytkownika

- Opisuje sposób korzystania z systemu
- Wyjaśnia dostępne funkcje aplikacji
- Zawiera przykłady wykonywania operacji
- Ułatwia użytkownikom pracę z systemem
- Może zawierać ilustracje i zrzuty ekranu



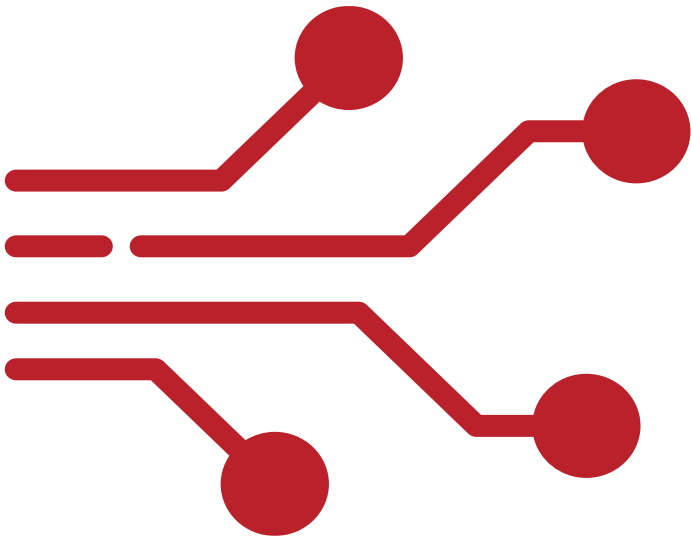
Dokumentacja administratora systemu

- Opisuje instalację i konfigurację systemu
- Zawiera informacje o zarządzaniu użytkownikami
- Dokumentuje parametry konfiguracyjne systemu
- Wyjaśnia procedury utrzymania i aktualizacji
- Wspiera zarządzanie infrastrukturą systemu



Dokumentacja instalacji i konfiguracji

- Opisuje proces instalacji systemu
- Określa wymagania sprzętowe i programowe
- Dokumentuje konfigurację środowiska
- Zawiera instrukcje uruchomienia systemu
- Wspiera wdrażanie systemu w nowych środowiskach



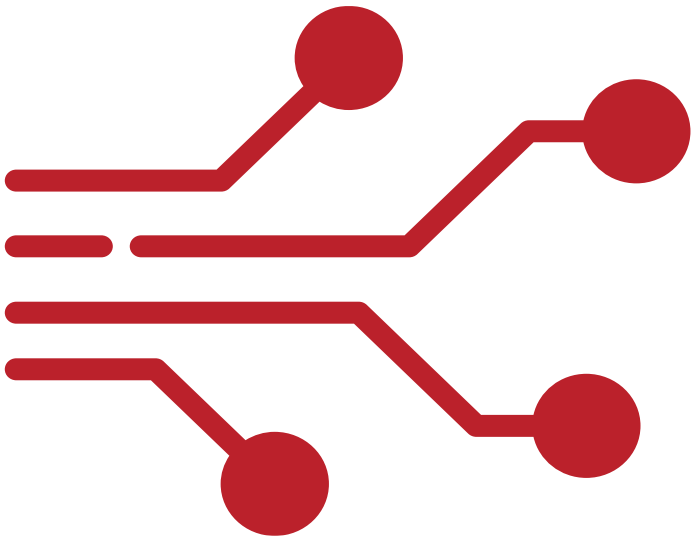
Dokumentacja w projektach open source

- Dokumentacja często znajduje się w repozytorium projektu
- Najczęściej wykorzystywany jest plik README
- Opisuje instalację, konfigurację i użycie systemu
- Może zawierać przykłady kodu
- Jest rozwijana przez społeczność projektu



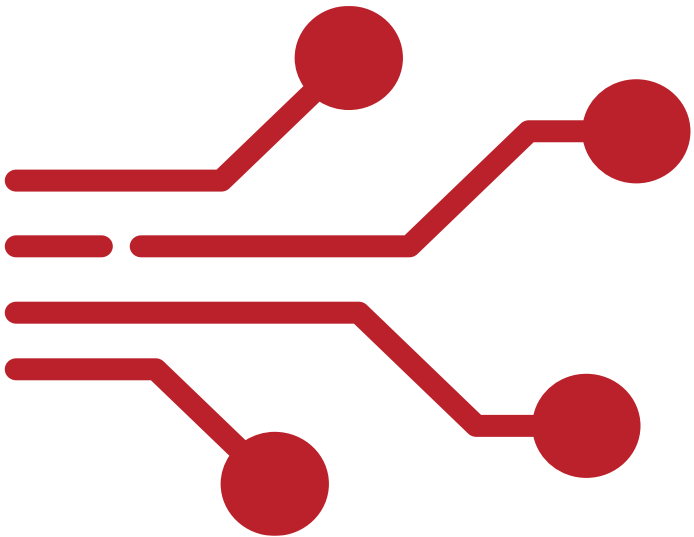
Komentarze w kodzie

- Wyjaśniają działanie fragmentów programu
- Opisują przeznaczenie klas i metod
- Ułatwiają zrozumienie kodu przez innych programistów
- Powinny być zwięzłe i aktualne
- Nie powinny powtarzać oczywistego działania kodu



Dokumentowanie klas i metod

- Opisuje przeznaczenie klasy
- Wyjaśnia działanie metod
- Dokumentuje parametry i wartości zwracane
- Ułatwia korzystanie z kodu przez innych programistów
- Może być automatycznie generowane przez narzędzia



Standardy dokumentowania kodu

- Określają sposób tworzenia komentarzy
- Definiują strukturę dokumentacji klas i metod
- Zapewniają spójność dokumentacji w projekcie
- Ułatwiają pracę zespołową
- Wspierają automatyczne generowanie dokumentacji



Narzędzia generujące dokumentację

- Generują dokumentację na podstawie kodu źródłowego
- Wykorzystują komentarze w kodzie
- Tworzą czytelne strony dokumentacji API
- Ułatwiają utrzymanie aktualnej dokumentacji
- Wspierają pracę zespołów programistycznych

Przykłady narzędzi:

- DocFX
- Doxygen
- Sandcastle
- Swagger/OpenAPI

Architektura systemu

- Określa strukturę systemu informatycznego
- Definiuje główne komponenty systemu
- Opisuje sposób komunikacji między modułami
- Wskazuje technologie wykorzystane w systemie
- Wspiera projektowanie i rozwój oprogramowania



331

Diagramy UML w dokumentacji architektury

- Przedstawiają strukturę systemu w formie graficznej
- Ułatwiają zrozumienie relacji między komponentami
- Wspierają projektowanie systemów informatycznych
- Pomagają dokumentować architekturę oprogramowania
- Ułatwiają komunikację w zespole projektowym

Wykorzystywane diagramy:

- Klas
- Komponentów
- Wdrożenia
- Sekwencji

ADR (Architecture Decision Records)

- Dokumentują decyzje architektoniczne w projekcie
- Wyjaśniają powody wyboru określonych rozwiązań
- Opisują rozważane alternatywy
- Ułatwiają zrozumienie architektury systemu
- Wspierają rozwój systemu w przyszłości



Dokumentowanie mechanizmów bezpieczeństwa

- Opisuje zastosowane zabezpieczenia systemu
- Dokumentuje mechanizmy uwierzytelniania i autoryzacji
- Przedstawia sposób ochrony danych
- Określa zasady zarządzania dostępem
- Wspiera analizę bezpieczeństwa systemu

Model zagrożeń systemu

- Identyfikuje potencjalne zagrożenia dla systemu
- Analizuje możliwe scenariusze ataków
- Wskazuje krytyczne elementy systemu
- Pomaga projektować mechanizmy zabezpieczeń
- Wspiera analizę a w systemie informatycznym



Dokumentacja konfiguracji bezpieczeństwa

- Opisuje ustawienia bezpieczeństwa systemu
- Dokumentuje konfigurację mechanizmów ochrony
- Określa parametry kontroli dostępu
- Zawiera informacje o szyfrowaniu i ochronie danych
- Wspiera utrzymanie bezpiecznej konfiguracji systemu



Dokumentowanie incydentów bezpieczeństwa

- Rejestruje zdarzenia naruszające bezpieczeństwo systemu
- Opisuje przebieg incydentu
- Dokumentuje przyczyny oraz skutki zdarzenia
- Wspiera analizę i zapobieganie przyszłym incydom
- Jest elementem zarządzania bezpieczeństwem systemu

337

Markdown i dokumentacja w repozytorium

- Dokumentacja może być przechowywana razem z kodem źródłowym
- Popularnym formatem dokumentacji jest Markdown
- Dokumentacja jest wersjonowana w systemie kontroli wersji
- Ułatwia współpracę w zespole programistycznym
- Zapewnia spójność między kodem a dokumentacją



Wiki projektowe

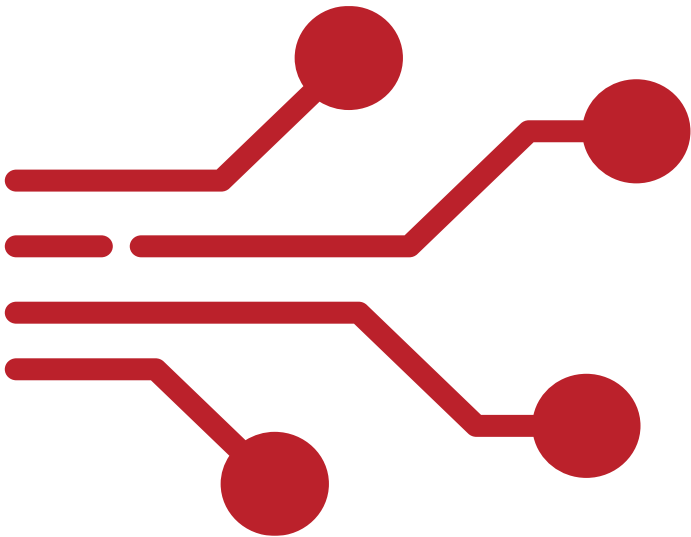
- Umożliwiają tworzenie wspólnej dokumentacji projektu
- Pozwalają łatwo edytować i aktualizować treści
- Ułatwiają współpracę w zespołach projektowych
- Mogą zawierać instrukcje, diagramy i procedury
- Wspierają zarządzanie wiedzą w projekcie

Przykłady systemów wiki:

- Confluence
- GitLab Wiki
- GitHub Wiki
- MediaWiki

Swagger/OpenAPI

- Narzędzia do dokumentowania interfejsów API
- Automatycznie generują dokumentację usług
- Tworzą interaktywną dokumentację API
- Ułatwiają testowanie usług sieciowych
- Wspierają integrację systemów informatycznych



Generatory dokumentacji

- Automatycznie tworzą dokumentację projektu
- Analizują kod źródłowy oraz komentarze
- Tworzą strony dokumentacji technicznej
- Ułatwiają utrzymanie aktualnej dokumentacji
- Wspierają rozwój projektów programistycznych

Przykłady generatorów dokumentacji:

- DocFX
- Doxygen
- Sphinx
- MkDocs

Cechy dobrej dokumentacji

- Jest zrozumiała dla odbiorców
- Jest aktualna i spójna z kodem
- Jest uporządkowana i czytelna
- Zawiera przykłady użycia systemu
- Jest łatwa do aktualizowania



Najczęstsze błędy w dokumentacji

- Dokumentacja jest nieaktualna
- Dokumentacja jest zbyt ogólna lub zbyt szczegółowa
- Brak spójności między dokumentacją a kodem
- Dokumentacja jest trudna do odnalezienia
- Brak przykładów użycia systemu



Dokumentacja w zespołach programistycznych

- Ułatwia komunikację między członkami zespołu
- Pozwala dzielić się wiedzą o systemie
- Wspiera wdrażanie nowych programistów
- Pomaga utrzymać spójność projektu
- Ogranicza o utraty wiedzy o systemie



Dokumentacja w DevOps

- Dokumentacja jest częścią procesu wytwarzania oprogramowania
- Wspiera automatyzację wdrażania systemu
- Opisuje konfigurację środowisk
- Dokumentuje proces budowania i wdrażania aplikacji
- Wspiera zarządzanie infrastrukturą systemu



345

Dokumentacja jako część repozytorium

- Dokumentacja jest przechowywana razem z kodem
- Podlega systemowi kontroli wersji
- Zmiany w dokumentacji są śledzone w historii projektu
- Ułatwia synchronizację dokumentacji z kodem
- Wspiera współpracę zespołu programistycznego



346

Dokumentacja a utrzymanie systemu

- Wspiera diagnozowanie problemów w systemie
- Ułatwia analizę błędów oraz awarii
- Pomaga wprowadzaniu zmian w systemie
- Ułatwia migrację systemów i aktualizacje
- Wspiera zarządzanie infrastrukturą informatyczną



Najważniejsze zasady dokumentowania

- Dokumentacja powinna być tworzona równolegle z kodem
- Powinna być dostosowana do odbiorców
- Powinna być aktualizowana wraz ze zmianami systemu
- Powinna być przechowywana w łatwo dostępnym miejscu
- Powinna wspierać rozwój i utrzymanie systemu

348

Dokumentacja a jakość oprogramowania

- Ułatwia zrozumienie architektury systemu
- Wspiera utrzymanie i rozwój oprogramowania
- Pomaga wykrywać błędy projektowe
- Ułatwia analizę bezpieczeństwa systemu
- Zwiększa niezawodność systemów informatycznych



Podsumowanie wykładu

- Dokumentacja jest ważnym elementem procesu wytwarzania oprogramowania
- Istnieją różne rodzaje dokumentacji systemów informatycznych
- Dokumentacja wspiera rozwój i utrzymanie systemu
- Nowoczesne projekty wykorzystują narzędzia automatyzujące dokumentację
- Dokumentacja odgrywa ważną rolę w bezpieczeństwie systemów

350

Dziękuję za uwagę



Wykład nr 8

Temat: Konserwacja oprogramowania

Paweł Karczmarek
Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

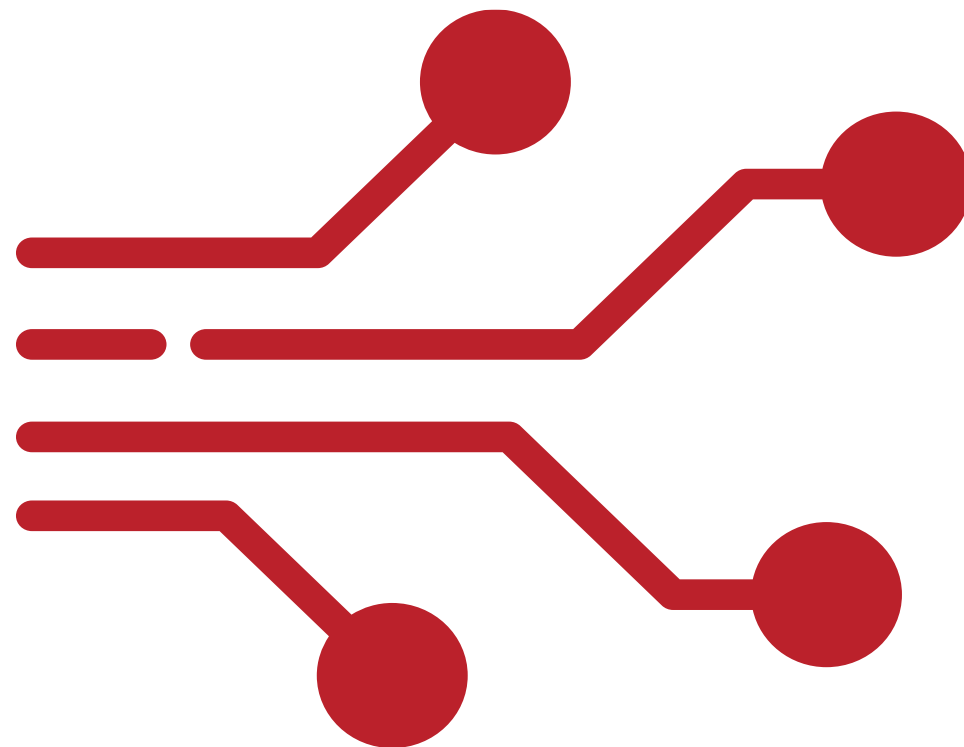
Plan prezentacji

- Wprowadzenie
- Typy konserwacji
- Refaktoryzacja



353

Wprowadzenie



354

Wprowadzenie

- Czym jest konserwacja oprogramowania?
- Typy konserwacji systemów
- Refaktoryzacja kodu
- Dług techniczny
- Zarządzanie zmianą w systemie



355

Dlaczego konserwacja oprogramowania jest istotna?

- Większość systemów działa przez wiele lat
- Wymagania biznesowe zmieniają się w czasie
- Środowisko technologiczne stale ewoluuje
- Użytkownicy zgłaszają błędy i nowe potrzeby



Koszt utrzymania systemów informatycznych

- W typowym cyklu życia systemu informatycznego
 - około 20-30% kosztów przypada na rozwój systemu
 - około 70-80% kosztów przypada na utrzymanie systemu

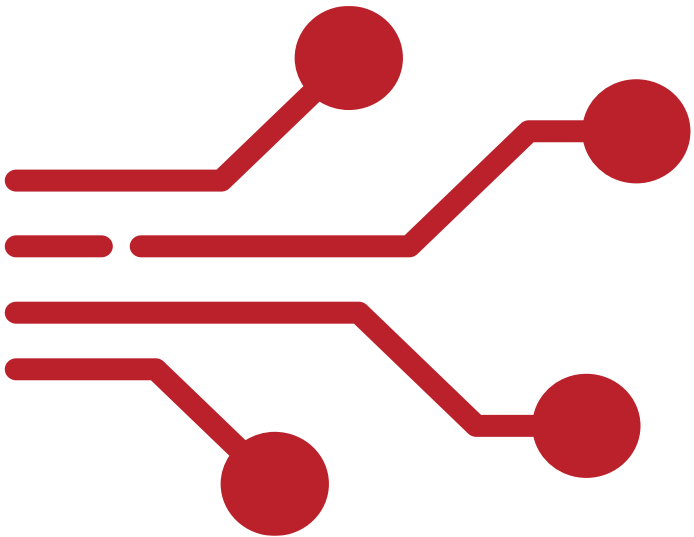
Koszty utrzymania obejmują między innymi

- Usuwanie błędów
- Dostosowywanie systemu do nowych warunków
- Rozwój funkcjonalności
- Aktualizacje technologiczne

357

Długowieczność systemów informatycznych

- Wysokie koszty stworzenia dużych systemów informatycznych
- Silna integracja systemów w infrastrukturze organizacji
- Zależność wielu procesów biznesowych od systemu
- Duża liczba użytkowników korzystających z systemu
- Ryzyko migracji danych oraz utraty spójności informacji



Konserwacja oprogramowania

- Modyfikacja oprogramowania po jego wdrożeniu i rozpoczęciu użytkowania
- Działania podejmowane w celu poprawy, dostosowania lub rozwoju systemu
- Usuwanie błędów wykrytych w trakcie eksploatacji
- Dostosowanie systemu do zmian technologicznych i organizacyjnych
- Rozszerzanie funkcjonalności systemu

359

Okres eksploatacji systemu informatycznego

- Rozpoczęcie użytkowania systemu po jego wdrożeniu
- Codzienna praca użytkowników z systemem
- Zgłaszanie błędów i problemów przez użytkowników
- Wprowadzanie poprawek i aktualizacji
- Stopniowy rozwój funkcjonalności systemu

360

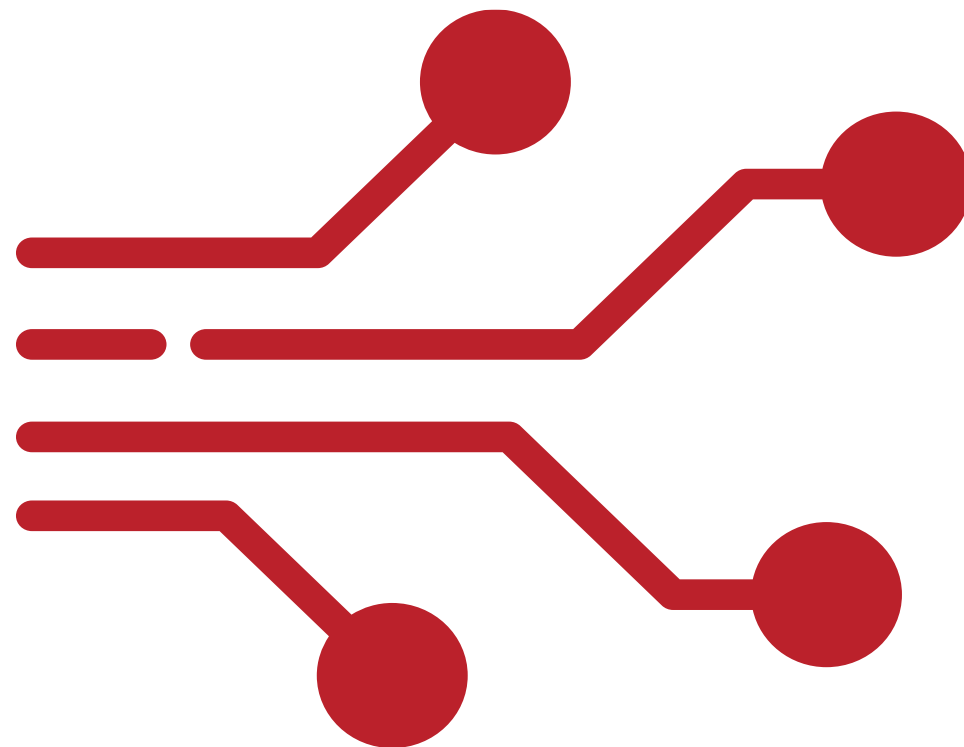
Rodzaje zmian w oprogramowaniu

- Usuwanie błędów wykrytych podczas eksploatacji systemu
- Dostosowywanie systemu do zmian technologicznych
- Dostosowywanie systemu do zmian organizacyjnych lub prawnych
- Rozszerzanie funkcjonalności systemu
- Poprawa wydajności i jakości działania systemu



361

Typy konserwacji



362

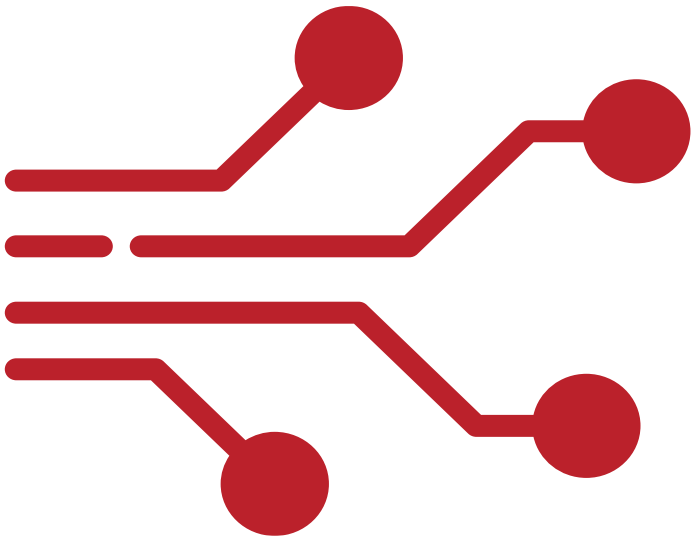
Typy konserwacji oprogramowania

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventive maintenance



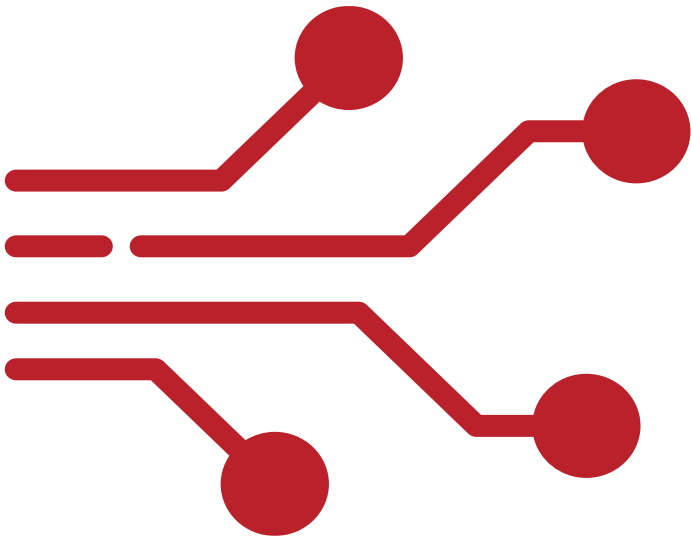
Corrective maintenance

- Usuwanie błędów wykrytych w trakcie użytkowania systemu
- Poprawa nieprawidłowego działania funkcji systemu
- Naprawa błędów logicznych w kodzie programu
- Usuwanie problemów związanych z bezpieczeństwem lub stabilnością systemu
- Poprawki publikowane w formie aktualizacji lub poprawek (patchy)



Adaptive maintenance

- Dostosowywanie systemu do zmian w środowisku technologicznym
- Aktualizacja oprogramowania do nowych wersji systemów operacyjnych
- Dostosowanie systemu do nowych wersji baz danych
- Zmiany wynikające z aktualizacji bibliotek i frameworków
- Integracja z nowymi systemami informatycznymi



365

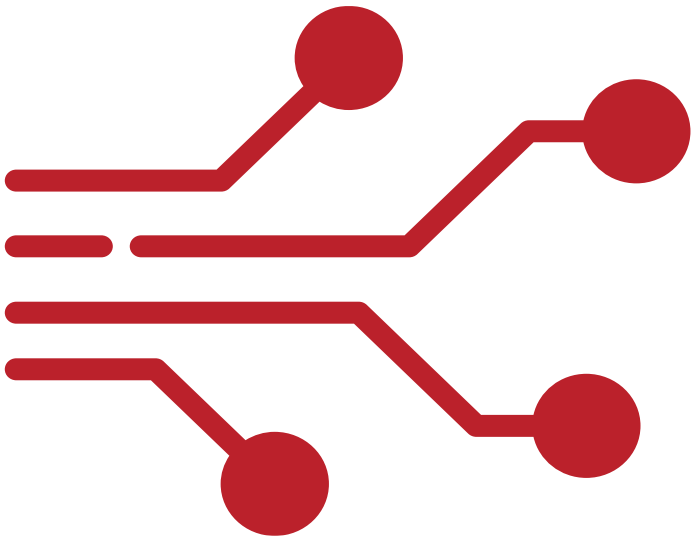
Perfective maintenance

- Rozwój i rozszerzanie funkcjonalności systemu
- Dostosowanie systemu do nowych potrzeb użytkowników
- Poprawa wydajności działania systemu
- Usprawnienie interfejsu użytkownika
- Optymalizacja działania istniejących funkcji



Preventive maintenance

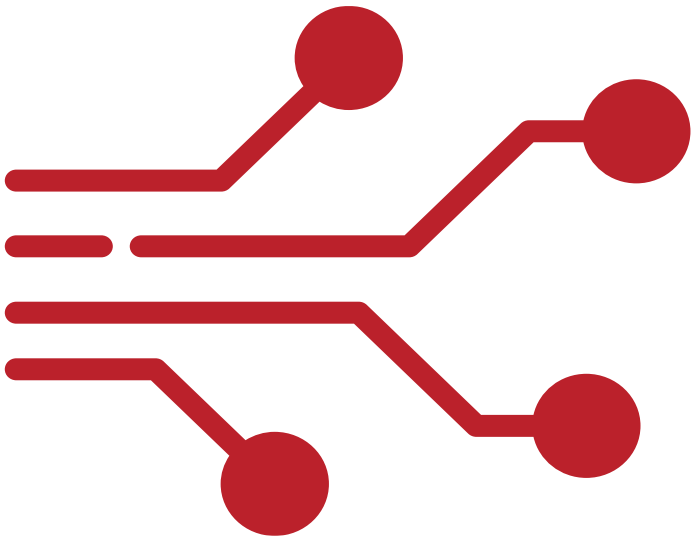
- Działania zapobiegające przyszłym problemom w systemie
- Poprawa struktury i czytelności kodu źródłowego
- Refaktoryzacja kodu programu
- Aktualizacja przestarzałych komponentów oprogramowania
- Zmniejszanie ryzyka pojawienia się błędów w przyszłości



367

Trudności w utrzymaniu systemów informatycznych

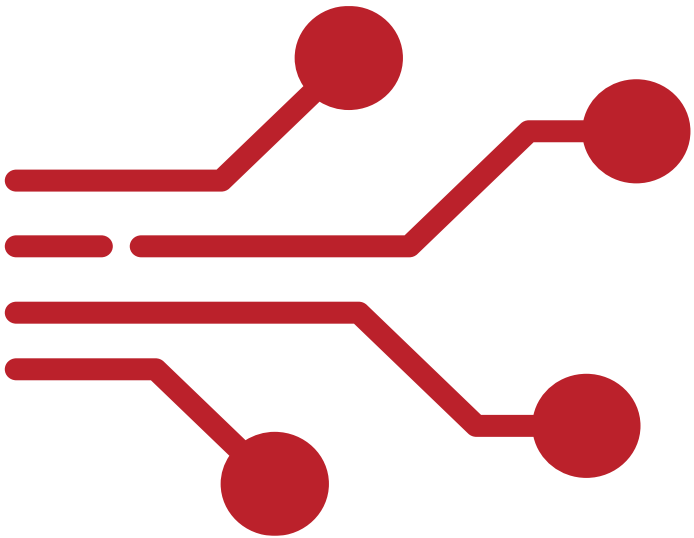
- Duża złożoność współczesnych systemów informatycznych
- Długi czas eksploatacji systemów
- Zmiany w środowisku technologicznym
- Zależności pomiędzy wieloma modułami systemu
- Konieczność zapewnienia ciągłości działania systemu



368

Systemy typu legacy

- Systemy informatyczne rozwijane i użytkowane przez wiele lat
- Oparte na starszych technologiach programistycznych
- Silnie zintegrowane z procesami biznesowymi organizacji
- Trudne w modernizacji lub zastąpieniu nowymi rozwiązaniami
- Nadal pełniące kluczową rolę w funkcjonowaniu organizacji



Problemy dokumentacji systemów

- Brak aktualnej dokumentacji systemu
- Dokumentacja niezgodna z rzeczywistą implementacją
- Brak dokumentacji architektury systemu
- Trudność w zrozumieniu zależności pomiędzy modułami
- Utrudnione wdrażanie nowych programistów do projektu



370

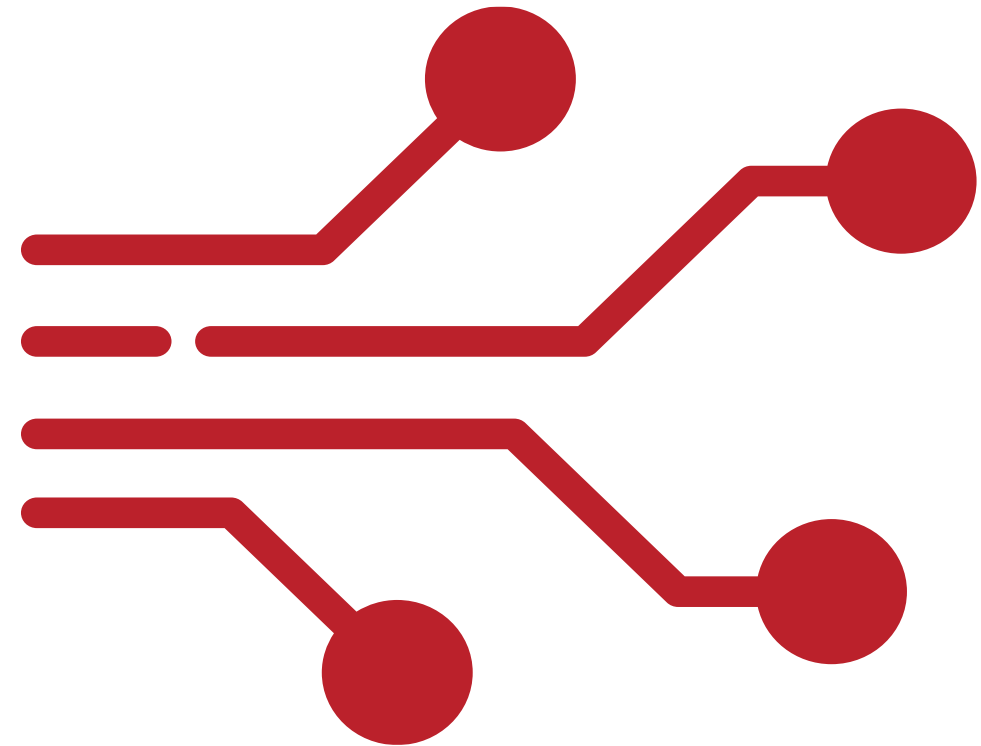
Rotacja programistów w projektach

- Zmiany w składzie zespołów programistycznych
- Odejścia doświadczonych programistów z projektu
- Utrata wiedzy o architekturze i implementacji systemu
- Konieczność wdrażania nowych członków zespołu
- o błędów podczas wprowadzania zmian w systemie



371

Refaktoryzacja



372

Refaktoryzacja kodu

- Zmiana struktury kodu bez zmiany jego funkcjonalności
- Poprawa czytelności i przejrzystości kodu źródłowego
- Uproszczenie struktury programu
- Usuwanie powielonego kodu
- Przygotowanie systemu do dalszego rozwoju



373

Przykłady refaktoryzacji

- Podział dużych funkcji na mniejsze metody
- Usuwanie powielonego kodu (duplicate code)
- Poprawa nazw zmiennych, metod i klas
- Wydzielanie fragmentów kodu do osobnych klas lub modułów
- Uproszczenie złożonych instrukcji warunkowych



374

Dług techniczny

- Konsekwencja szybkiego lub nieoptymalnego tworzenia oprogramowania
- Kompromis pomiędzy szybkością implementacji a jakością kodu
- Nagromadzenie problemów w strukturze systemu
- Utrudnienie dalszego rozwoju oprogramowania
- Wzrost kosztów utrzymania systemu



375

Powstawanie długu technicznego

- Presja czasu podczas realizacji projektu
- Brak refaktoryzacji kodu w kolejnych etapach rozwoju systemu
- Wprowadzanie szybkich rozwiązań tymczasowych
- Brak spójnych standardów programistycznych w projekcie
- Niedostateczne testowanie oprogramowania



376

Konsekwencje długu technicznego

- Spowolnienie rozwoju systemu
- Zwiększenie kosztów utrzymania oprogramowania
- Więcej wprowadzania błędów przy zmianach w systemie
- Trudności w zrozumieniu kodu przez nowych programistów
- Większa podatność systemu na problemy bezpieczeństwa



377

Zarządzanie zmianą w systemie informatycznym

- Formalny proces wprowadzania zmian w systemie
- Rejestrowanie propozycji zmian w systemie zgłoszeń
- Analiza wpływu zmiany na działanie systemu
- Planowanie wdrożenia zmiany
- Testowanie i wdrożenie zmodyfikowanej wersji systemu



378

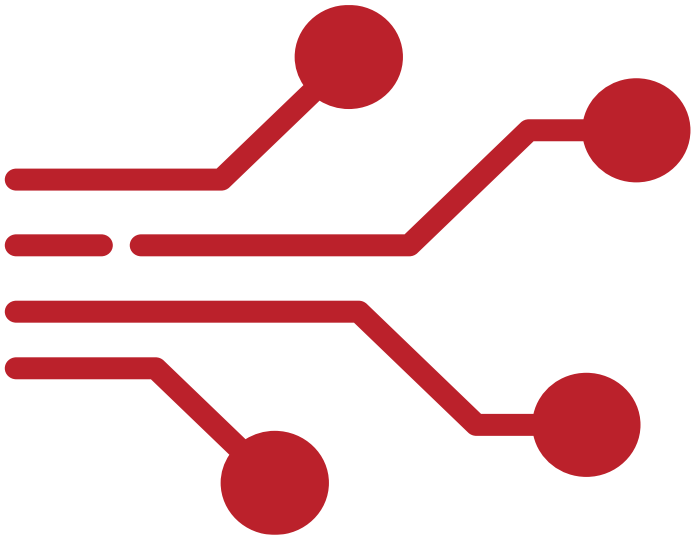
Analiza wpływu zmiany

- Identyfikacja elementów systemu objętych zmianą
- Analiza zależności pomiędzy modułami systemu
- Ocena wpływu zmiany na inne funkcje systemu
- Oszacowanie kosztu i czasu implementacji zmiany
- Identyfikacja potencjalnych związanych z wprowadzeniem zmiany



Priorytetyzacja zmian

- Określenie ważności zgłoszonych zmian
- Uwzględnienie wpływu zmiany na działanie systemu
- Ocena znaczenia zmiany dla procesów biznesowych
- Uwzględnienie dostępnych zasobów zespołu projektowego
- Planowanie realizacji zmian w kolejnych wersjach systemu



380

Narzędzia wspierające utrzymanie systemów

- Systemy zarządzania zgłoszeniami i błędami (issue tracking systems)
- Systemy kontroli wersji oprogramowania
- Narzędzia do monitorowania działania systemów
- Narzędzia do automatycznego testowania oprogramowania
- Narzędzia do analizy jakości kodu



Monitorowanie systemów informatycznych

- Obserwacja działania systemu w czasie rzeczywistym
- Monitorowanie wydajności serwerów i aplikacji
- Analiza wykorzystania zasobów systemowych
- Wykrywanie błędów i nieprawidłowości w działaniu systemu
- Rejestrowanie zdarzeń w systemach logowania



Automatyzacja utrzymania systemów

- Automatyzacja testów oprogramowania
- Automatyczne wdrażanie nowych wersji systemu
- Automatyczne monitorowanie działania aplikacji
- Automatyczne tworzenie kopii zapasowych danych
- Wykorzystanie narzędzi CI/CD w procesie utrzymania systemu



Dobre praktyki utrzymania oprogramowania

- Regularna refaktoryzacja kodu źródłowego
- Utrzymywanie aktualnej dokumentacji systemu
- Stosowanie automatycznych testów oprogramowania
- Regularne aktualizowanie bibliotek i komponentów systemu
- Monitorowanie działania systemu oraz analiza logów



Aktualizacje bezpieczeństwa i patch management

- Instalowanie poprawek usuwających podatności bezpieczeństwa
- Aktualizowanie systemów operacyjnych, bibliotek i aplikacji
- Szybka reakcja na wykryte luki bezpieczeństwa
- Planowanie cykli aktualizacji w środowisku produkcyjnym
- Ograniczanie a wykorzystania znanych podatności

385

CVE i zarządzanie podatnościami

- CVE (Common Vulnerabilities and Exposures) - publiczna baza podatności
- Identyfikacja znanych luk bezpieczeństwa w oprogramowaniu
- Przypisywanie unikalnych identyfikatorów podatności
- Monitorowanie nowych podatności w używanym oprogramowaniu
- Planowanie aktualizacji eliminujących podatności

386

Wersje LTS (Long Term Support)

- Wersje oprogramowania objęte długoterminowym wsparciem
- Regularne aktualizacje bezpieczeństwa przez wiele lat
- Stabilność systemu w środowiskach produkcyjnych
- Ograniczenie częstotliwości dużych zmian technologicznych
- Stosowane w systemach o wysokiej niezawodności



387

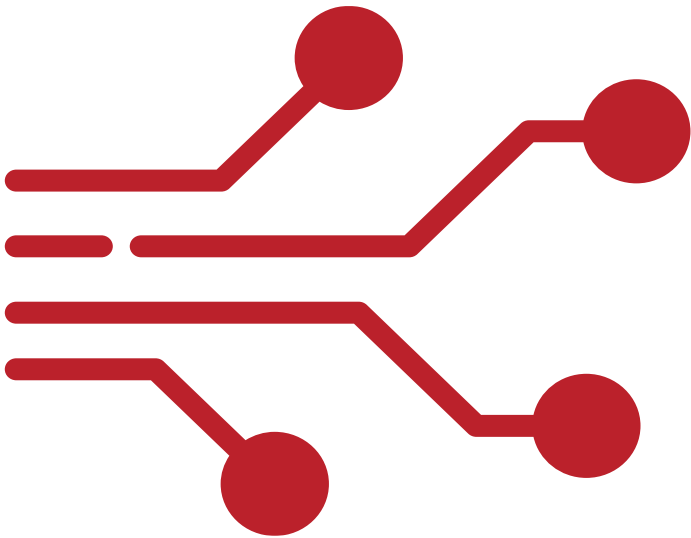
Reagowanie na incydenty bezpieczeństwa

- Identyfikacja incydentów bezpieczeństwa w systemie
- Analiza przyczyn zdarzenia
- Ograniczenie skutków incydentu
- Usunięcie podatności wykorzystanej w ataku
- Wprowadzenie działań zapobiegających podobnym zdarzeniom



Podsumowanie

- Konserwacja oprogramowania stanowi istotną część cyklu życia systemu
- Większość kosztów systemu informatycznego związana jest z jego utrzymaniem
- W utrzymaniu systemów stosuje się różne typy konserwacji
- Refaktoryzacja i ograniczanie długu technicznego ułatwiają rozwój systemu
- Formalne procesy zarządzania zmianą zwiększają stabilność systemu



Dziękuję za uwagę

Wykład nr 9

Temat: Systemy krytyczne

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

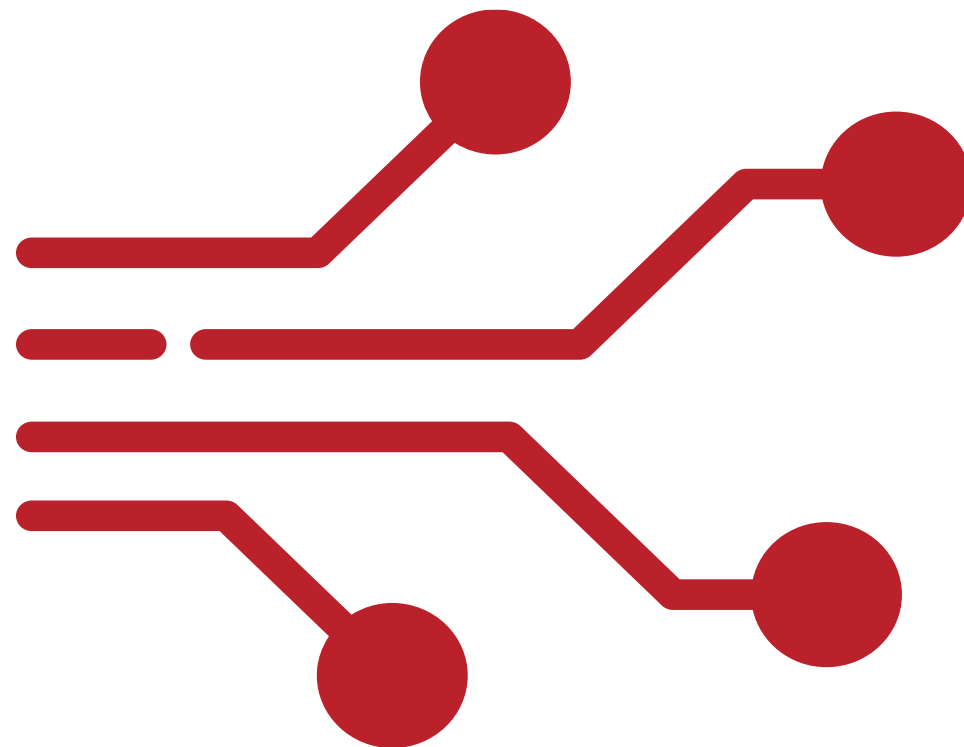
Plan prezentacji

- Wprowadzenie
- Rodzaje systemów krytycznych i procedury



392

Wprowadzenie



393

Czym są systemy krytyczne?

- Systemy, których awaria może prowadzić do poważnych skutków
- Mogą wpływać na życie i zdrowie ludzi
- Mogą powodować straty finansowe lub środowiskowe
- Są wykorzystywane w obszarach o wysokim poziomie odpowiedzialności
- Wymagają wysokiej niezawodności, bezpieczeństwa i kontroli jakości



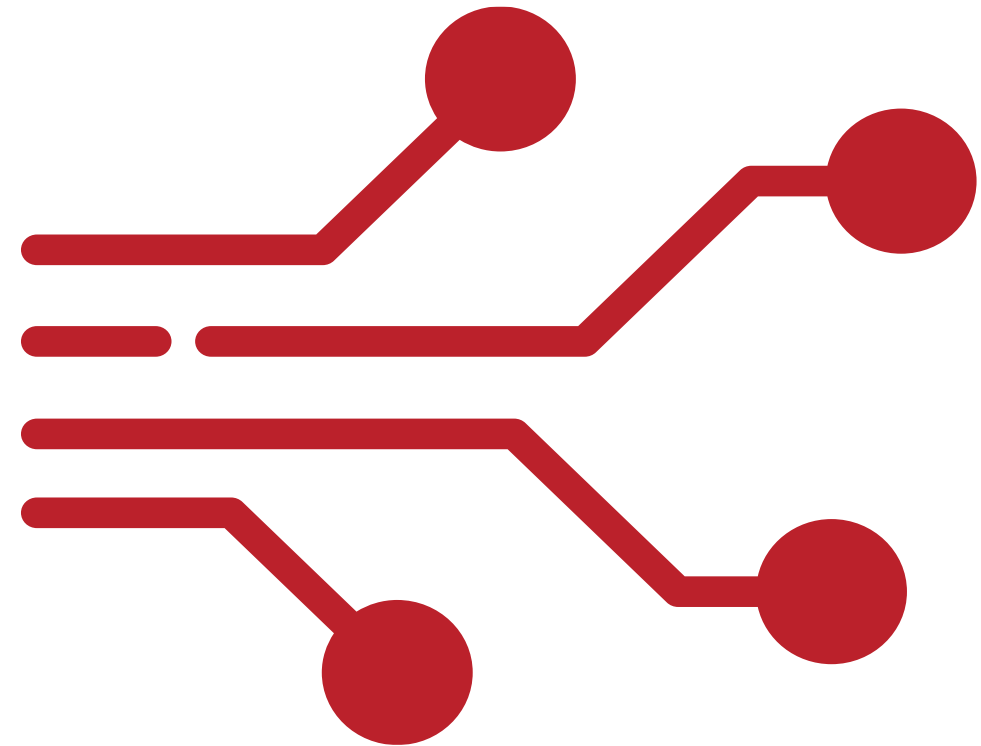
394

Dlaczego awaria systemu krytycznego jest szczególnie niebezpieczna?

- Może prowadzić do zagrożenia życia i zdrowia ludzi
- Może powodować poważne straty finansowe
- Może zakłócić działanie infrastruktury publicznej
- Może prowadzić do katastrof technicznych lub środowiskowych
- Może mieć konsekwencje społeczne i prawne



Rodzaje systemów krytycznych i procedury



396

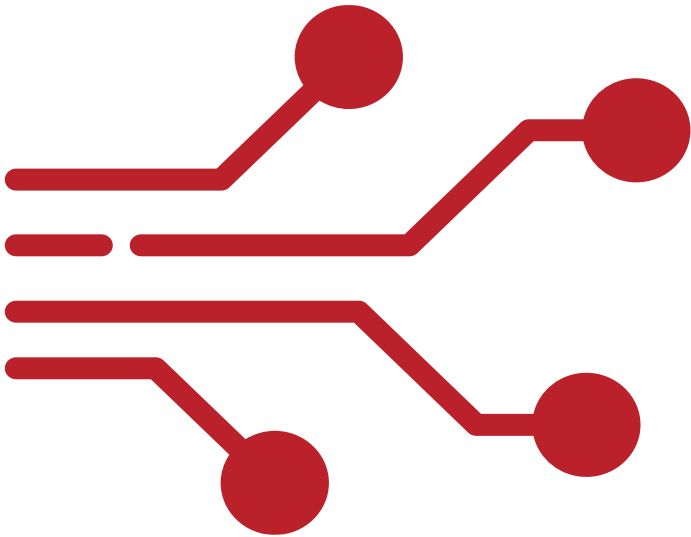
Przykłady systemów krytycznych

- Systemy sterowania ruchem lotniczym
- Systemy sterowania ruchem kolejowym
- Urządzenia medyczne i systemy szpitalne
- Systemy sterujące elektrowniami i sieciami energetycznymi
- Systemy sterowania procesami przemysłowymi



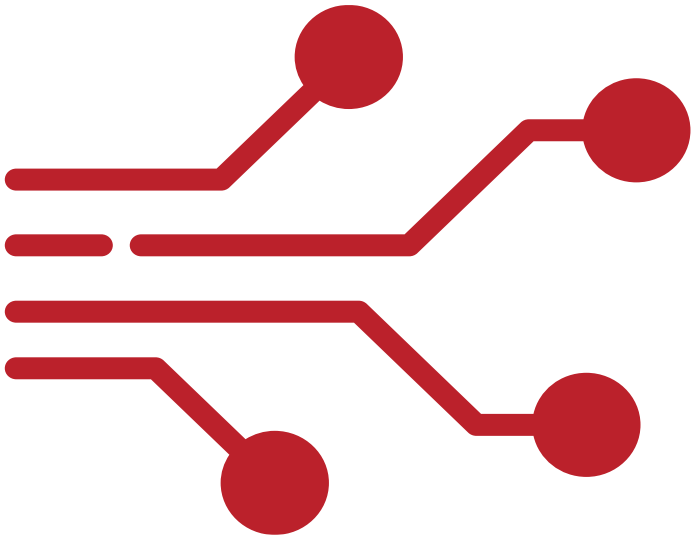
Systemy krytyczne a zwykłe systemy informatyczne

- Awaria zwykłego systemu oznacza najczęściej utrudnienia dla użytkowników
- Awaria systemu krytycznego może prowadzić do poważnych konsekwencji
- Systemy krytyczne wymagają wyższych standardów jakości i testowania
- Proces projektowania systemów krytycznych jest bardziej rygorystyczny
- Systemy krytyczne podlegają często certyfikacji i regulacjom



Systemy krytyczne ze względu na bezpieczeństwo ludzi

- Awaria może bezpośrednio zagrozić życiu lub zdrowiu ludzi
- Systemy sterujące urządzeniami medycznymi
- Systemy sterowania ruchem lotniczym i kolejowym
- Systemy bezpieczeństwa w pojazdach i przemyśle
- Systemy kontroli procesów w instalacjach przemysłowych



399

Systemy krytyczne ze względu na niezawodność działania

- Awaria powoduje przerwanie działania kluczowych usług
- Systemy infrastruktury energetycznej
- Systemy telekomunikacyjne
- Systemy bankowe i finansowe
- Systemy obsługujące infrastrukturę publiczną



Systemy krytyczne ze względu na poufność i integralność danych

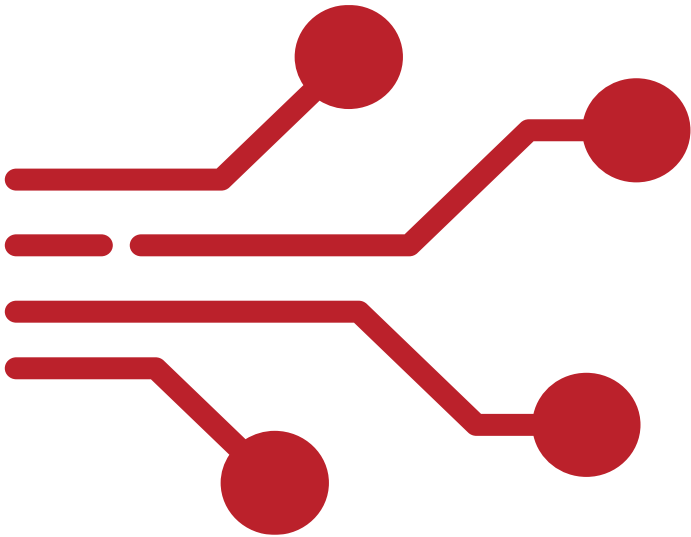
- Chronią wrażliwe lub strategiczne informacje
- Wymagają wysokiego poziomu bezpieczeństwa informacji
- Systemy bankowe i finansowe
- Systemy administracji publicznej
- Systemy wojskowe i systemy bezpieczeństwa państwa



401

Systemy czasu rzeczywistego jako szczególny typ systemów krytycznych

- Systemy, które muszą reagować w określonym czasie
- Opóźnienie reakcji może prowadzić do błędnego działania systemu
- Wykorzystywane w sterowaniu urządzeniami i procesami
- Stosowane w lotnictwie, motoryzacji i przemyśle
- Wymagają przewidywalnego czasu działania oprogramowania



Systemy krytyczne w lotnictwie

- Systemy sterowania lotem samolotu
- Systemy sterowania ruchem lotniczym
- Systemy nawigacji i autopilota
- Systemy monitorowania parametrów lotu
- Systemy wspomagające bezpieczeństwo lotu



403

Systemy krytyczne w medycynie

- Urządzenia medyczne sterowane oprogramowaniem
- Systemy monitorowania pacjentów
- Pompy insulinowe i urządzenia terapeutyczne
- Systemy radioterapii i diagnostyki medycznej
- Systemy informatyczne zarządzające danymi pacjentów



Systemy krytyczne w energetyce i przemyśle

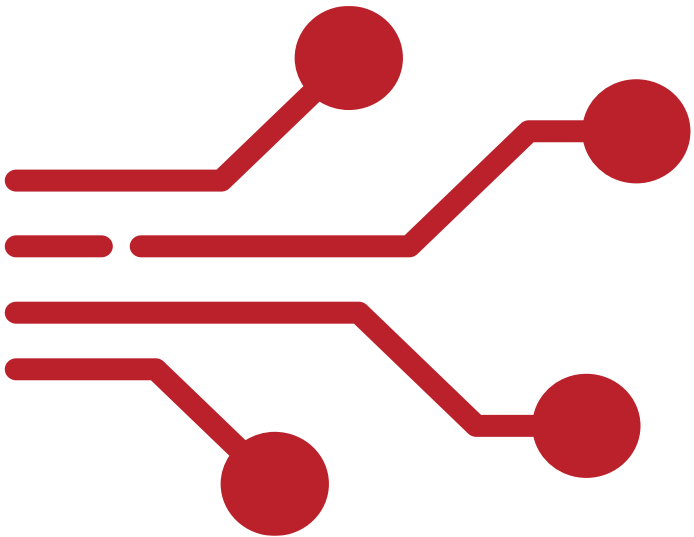
- Systemy sterujące elektrowniami
- Systemy zarządzania sieciami energetycznymi
- Systemy SCADA w przemyśle
- Systemy sterowania procesami produkcyjnymi
- Systemy monitorowania infrastruktury technicznej



405

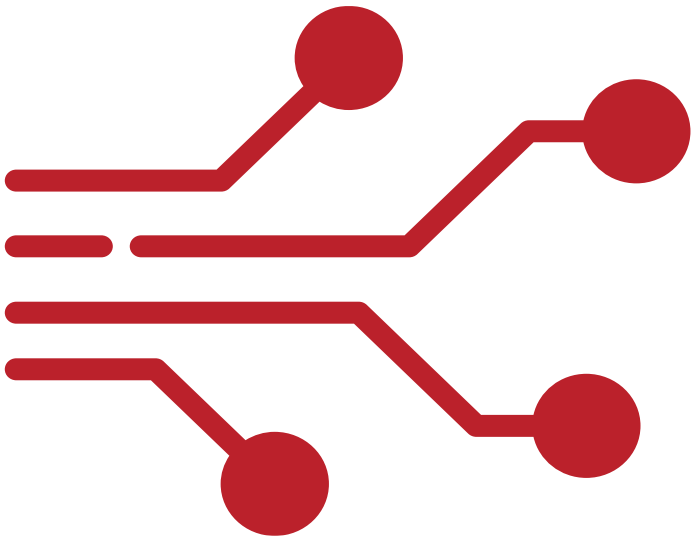
SCADA (Supervisory Control And Data Acquisition)

- Systemy nadzoru i sterowania procesami przemysłowymi
- Monitorowanie, zbieranie danych i automatyczne sterowanie
- Wykorzystywane w energetyce, wodociągach, rafineriach, produkcji



Systemy krytyczne w transporcie i infrastrukturze publicznej

- Systemy sterowania ruchem kolejowym
- Systemy zarządzania ruchem drogowym
- Systemy metra i transportu miejskiego
- Systemy zarządzania lotniskami i portami
- Systemy infrastruktury miejskiej



407

Niezawodność systemu

- Zdolność systemu do poprawnego działania przez określony czas
- Minimalizacja liczby błędów i awarii
- Odporność na błędy sprzętowe i programowe
- Projektowanie systemów z myślą o ciągłości działania
- Kluczowa cecha systemów krytycznych



Dostępność systemu

- Zdolność systemu do ciągłego świadczenia usług
- Minimalizacja przestojów i przerw w działaniu
- System powinien być dostępny wtedy, gdy jest potrzebny
- Stosowanie mechanizmów wysokiej dostępności (High Availability)
- Wykorzystywanie redundancji i mechanizmów przełączania awaryjnego



Bezpieczeństwo funkcjonalne (functional safety)

- Zapewnienie, że system działa poprawnie w obecności błędów
- Minimalizowanie a niebezpiecznych skutków awarii
- Stosowane w systemach sterowania procesami, pojazdach, urządzeniach medycznych
- Zgodność z normami bezpieczeństwa (np. ISO 26262, IEC 61508)
- Wymaga analizy ryzyka, redundancji i testów awaryjnych

410

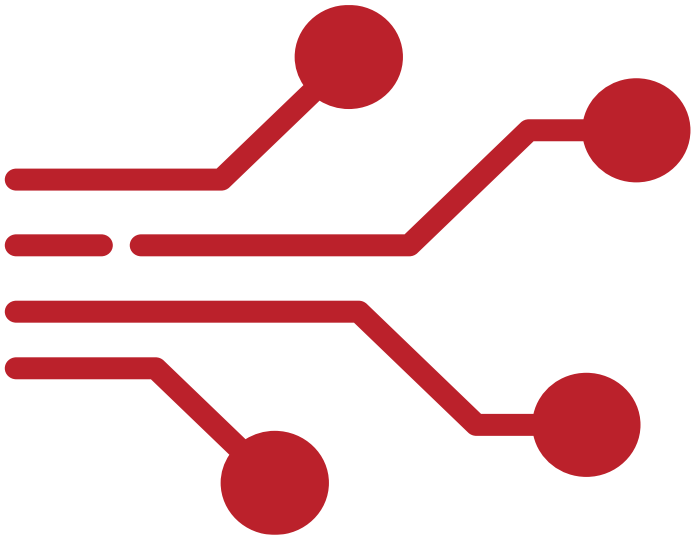
Poziomy SIL (Safety Integrity Level)

- Safety Integrity Level, ocena bezpieczeństwa funkcjonalnego
- SIL1: dopuszczalna częstość awarii: 10^{-5} - 10^{-6} /h
- SIL2: 10^{-6} - 10^{-7} /h
- SIL3: 10^{-7} - 10^{-8} /h
- SIL4: 10^{-8} - 10^{-9} /h
- Wyższy SIL oznacza wyższe wymagania projektowe i testowe
- Wartości są podane dla pracy ciągłej

411

Odporność na błędy i awarie (fault tolerance)

- System działa poprawnie nawet przy awarii części komponentów
- Wykrywanie błędów i automatyczne reagowanie
- Redundancja sprzętowa i programowa
- Mechanizmy fail-safe i failover
- Minimalizowanie skutków awarii dla użytkownika i środowiska



412

Źródła błędów w systemach krytycznych

- Błędy programistyczne i logiczne w kodzie
- Awaria sprzętu lub komponentów infrastruktury
- Błędy w projektowaniu systemu i architekturze
- Nieprawidłowa konfiguracja i zarządzanie systemem
- Czynniki zewnętrzne: środowisko, warunki fizyczne, ataki cybernetyczne



Skutki awarii systemów krytycznych

- Zagrożenie życia i zdrowia ludzi
- Przerwy w działaniu infrastruktury i usług publicznych
- Straty finansowe dla organizacji i państwa
- Uszkodzenia środowiska lub mienia
- Utrata zaufania i konsekwencje prawne



414

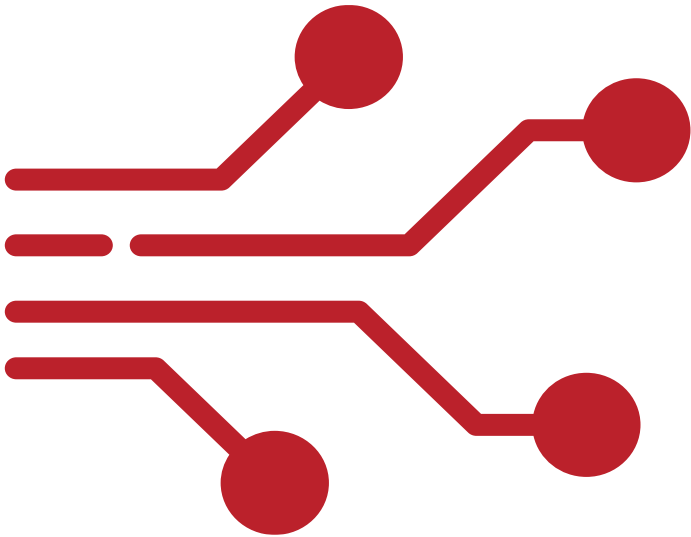
Błędy programistyczne a katastrofy techniczne

- Nieprawidłowy kod może prowadzić do awarii systemu
- Błędy w oprogramowaniu sterującym urządzeniami krytycznymi
- Przykłady: samoloty, elektrownie, urządzenia medyczne
- Małe błędy mogą mieć poważne konsekwencje w systemach krytycznych
- Właściwe testowanie i walidacja minimalizują o katastrof

415

Analiza a w systemach krytycznych

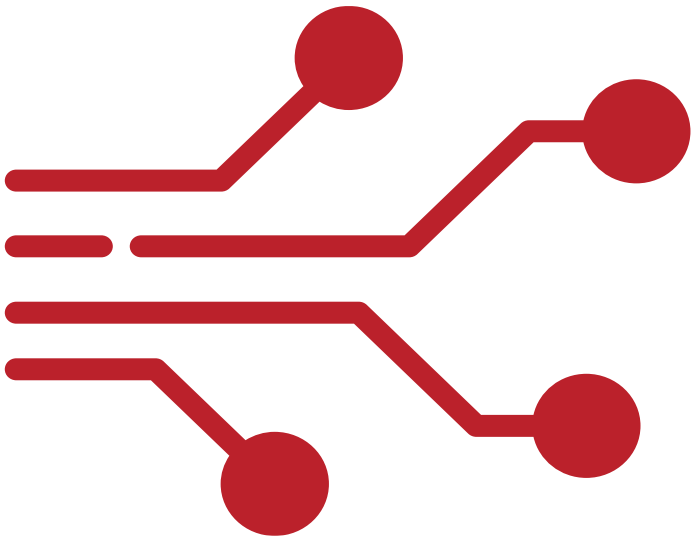
- Identyfikacja potencjalnych zagrożeń dla systemu
- Ocena prawdopodobieństwa wystąpienia awarii
- Określenie skutków awarii dla ludzi, infrastruktury i środowiska
- Priorytetyzacja działań zapobiegawczych
- Stosowanie procedur minimalizujących ryzyko



416

HAZOP (Hazard and Operability Study)

- Systematyczna analiza zagrożeń i problemów operacyjnych
- Identyfikacja potencjalnych awarii, ich przyczyn i skutków
- Ocena ryzyka i propozycje działań korygujących
- Stosowana w przemyśle chemicznym, energetycznym i procesowym



417

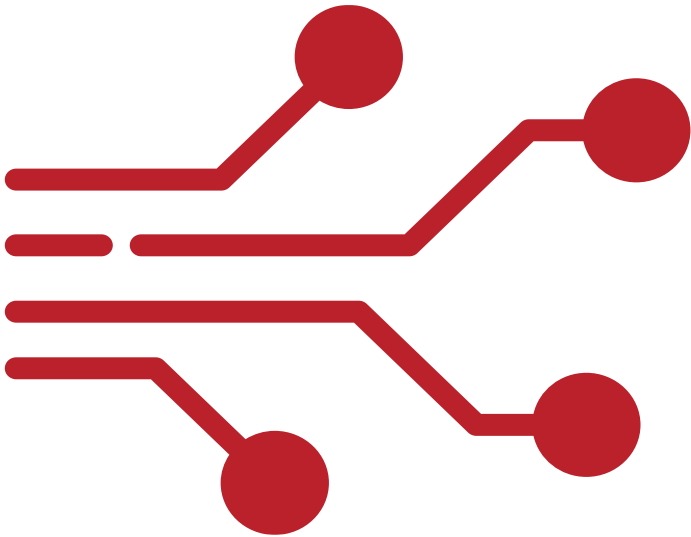
Wymagania dla systemów krytycznych

- Specyfikacja funkcjonalna i нефunkcjonalna
- Wysokie standardy niezawodności i dostępności
- Wymagania bezpieczeństwa funkcjonalnego
- Zgodność z normami branżowymi i regulacjami prawnymi
- Dokładna dokumentacja i procedury weryfikacyjne



Redundancja sprzętowa i programowa

- Zapasowe komponenty przejmują działanie w przypadku awarii
- Redundancja sprzętowa: serwery, czujniki, urządzenia
- Redundancja programowa: alternatywne algorytmy i procedury
- Zwiększa niezawodność i dostępność systemu
- Minimalizuje skutki awarii dla użytkowników i procesów



419

Mechanizmy fail-safe i fault-tolerant

- Fail-safe - system przechodzi w bezpieczny stan w przypadku awarii
- Fault-tolerant - system kontynuuje działanie mimo błędów
- Wykorzystywane w lotnictwie, energetyce, medycynie i przemyśle
- Minimalizują o katastrofy w sytuacjach awaryjnych
- Często łączone z redundancją sprzętową i programową



420

Projektowanie z myślą o odporności na błędy

- Minimalizacja wpływu awarii na działanie systemu
- Analiza krytycznych komponentów i punktów awarii
- Projektowanie redundancji i mechanizmów wykrywania błędów
- Stosowanie monitorowania stanu systemu i automatycznych reakcji
- Zapewnienie bezpiecznego zachowania w sytuacjach awaryjnych



421

Znaczenie testowania w systemach krytycznych

- Weryfikacja poprawności działania wszystkich funkcji
- Testowanie odporności na awarie i błędy
- Symulacja scenariuszy krytycznych
- Zapewnienie zgodności z normami i regulacjami
- Minimalizacja a awarii w warunkach rzeczywistych



422

Testy funkcjonalne i нефункционалне

- Testy funkcjonalne - sprawdzają, czy system realizuje określone funkcje
- Testy нефункционалне - sprawdzają niezawodność, wydajność i dostępność
- Testy odporności na awarie i scenariusze krytyczne
- Testy bezpieczeństwa i integralności danych
- Formalne procedury i dokumentacja wyników testów

423

Walidacja, certyfikacja i zgodność z normami

- Potwierdzenie, że system spełnia wymagania projektowe i bezpieczeństwa
- Certyfikacja przez niezależne jednostki w branży
- Zgodność z normami: ISO 26262, IEC 61508, DO-178C, EN 50128
- Dokumentacja procesu walidacji i wyników testów
- Nieodzowny element dopuszczenia systemu do eksploatacji

424

Formalne metody w systemach krytycznych

- Matematyczne techniki projektowania i weryfikacji systemu
- Specyfikacja formalna: opis funkcji i wymagań bez niejednoznaczności
- Weryfikacja poprawności działania algorytmów i procedur
- Dowody poprawności i symulacje scenariuszy krytycznych
- Stosowane w lotnictwie, motoryzacji, energetyce i medycynie

425

Normy oprogramowania dla systemów krytycznych

- IEC 61508 - standard bezpieczeństwa funkcjonalnego dla systemów przemysłowych
- ISO 26262 - standard dla systemów elektrycznych i elektronicznych w motoryzacji
- Określają procesy projektowania, testowania, weryfikacji i certyfikacji systemów
- Wymagają dokumentowania wymagań, analizy a i procedur walidacji
- Podstawa dla certyfikacji i zgodności z SIL

426

Zagrożenia cyberbezpieczeństwa dla systemów krytycznych

- Nieautoryzowany dostęp do systemu
- Ataki na integralność i poufność danych
- Manipulacja działaniem systemu lub jego sterowaniem
- Wirusy, malware, ransomware w systemach krytycznych
- Ataki z wykorzystaniem sieci i Internetu rzeczy (IoT)



427

Ataki na infrastrukturę krytyczną

- Ataki DDoS i zakłócenia usług sieciowych
- Włamania do systemów sterowania i SCADA
- Sabotaż cyberfizyczny - manipulacja urządzeniami i procesami
- Ataki na systemy IoT i urządzenia zdalne
- Kradzież danych i wyłudzenia w systemach finansowych i energetycznych

428

Bezpieczna architektura systemów krytycznych

- Segmentacja i izolacja sieci
- Kontrola dostępu i uwierzytelnianie wieloskładnikowe
- Szyfrowanie danych w spoczynku i w transmisji
- Redundancja i mechanizmy fail-safe/fault-tolerant
- Monitorowanie i audyt zdarzeń bezpieczeństwa
- Stosowanie zasad minimalnych uprawnień (least privilege)



429

Monitorowanie, detekcja incydentów i reagowanie

- Ciągłe monitorowanie stanu systemu i sieci
- Wykrywanie nieprawidłowości i anomalii
- Alertowanie i automatyczne reakcje na zagrożenia
- Procedury reagowania na incydenty bezpieczeństwa
- Analiza i raportowanie w celu doskonalenia systemu



430

Systemy bezpieczeństwa fizycznego

- SSWiN - System Sygnalizacji Włamania i Napadu
- SSP - System Sygnalizacji Pożaru
- BMS - Building Management System (zarządzanie budynkiem)
- CCTV - Closed-Circuit Television (monitoring wizyjny)
- KD - Kontrola Dostępu
- Perymetria - monitoring stref zewnętrznych (ogrodzenia, bramy, strefy ochronne)
- Integracja w celu pełnej kontroli bezpieczeństwa
- Centralizacja danych i zdarzeń w PSIM - Physical Security Information Management

431

Systemy bezpieczeństwa fizycznego

- Monitorowanie i ochrona infrastruktury krytycznej
- Szybkie wykrywanie zagrożeń i sytuacji awaryjnych
- Podstawa do integracji z innymi systemami bezpieczeństwa



432

Integracja i znaczenie PSIM (Physical Security Information Management)

- Integruje wszystkie systemy bezpieczeństwa fizycznego w jedną platformę
- Umożliwia korelację zdarzeń i automatyczne reakcje
- Poprawia skuteczność monitorowania i reagowania
- Kluczowe w systemach krytycznych i infrastrukturze wrażliwej

433

Konserwacja i aktualizacje systemów krytycznych

- Regularne przeglądy i serwisowanie komponentów sprzętowych
- Aktualizacje oprogramowania i firmware'u
- Testowanie po aktualizacjach przed wdrożeniem
- Zarządzanie cyklem życia komponentów systemu
- Minimalizacja a awarii i luk bezpieczeństwa



434

Zarządzanie zmianą w systemach krytycznych

- Formalne procedury wprowadzania zmian w systemie
- Analiza wpływu zmiany na niezawodność i bezpieczeństwo
- Testowanie i weryfikacja zmian przed wdrożeniem
- Dokumentacja wszystkich modyfikacji i decyzji
- Minimalizacja a niepożądanych skutków zmian

435

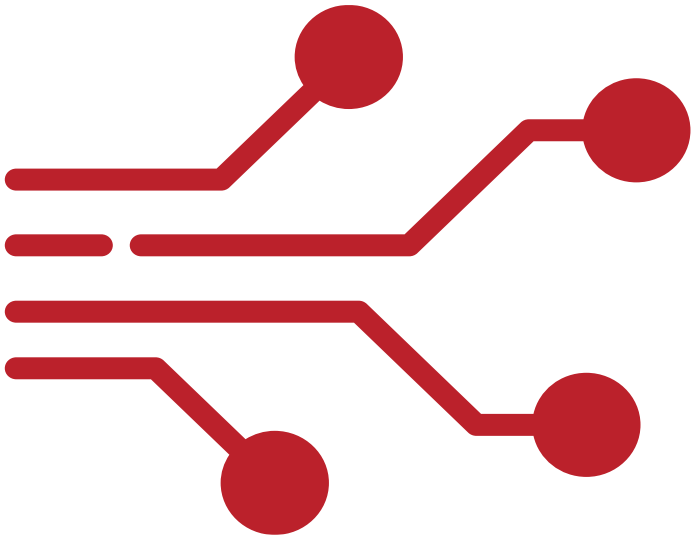
Dokumentacja i procedury operacyjne

- Instrukcje obsługi systemu i procedury awaryjne
- Dokumentacja konfiguracji i zależności systemowych
- Rejestracja zdarzeń, incydentów i działań naprawczych
- Procedury monitorowania, konserwacji i aktualizacji
- Podstawa szkoleń personelu i audytów bezpieczeństwa

436

Najważniejsze zasady projektowania systemów krytycznych

- Projektowanie z myślą o bezpieczeństwie ludzi i środowiska
- Wysoka niezawodność, dostępność i odporność na awarie
- Redundancja, mechanizmy fail-safe i fault-tolerant
- Formalna weryfikacja, testy i analiza ryzyka
- Dokumentacja, procedury operacyjne i zgodność z normami



437

Systemy krytyczne a odpowiedzialność inżynierska

- Projektowanie i wdrażanie zgodnie z zasadami etyki i prawa
- Odpowiedzialność za bezpieczeństwo ludzi, środowiska i infrastruktury
- Dokładne dokumentowanie decyzji projektowych i testów
- Zgłaszanie i reagowanie na potencjalne zagrożenia
- Stałe doskonalenie wiedzy i umiejętności w zakresie bezpieczeństwa

438

Podsumowanie

- Systemy krytyczne są kluczowe dla bezpieczeństwa, życia i infrastruktury
- Wysoka niezawodność, dostępność i odporność na błędy
- Analiza ryzyka, redundancja, mechanizmy fail-safe/fault-tolerant
- Formalna weryfikacja, testy, walidacja i certyfikacja
- Cyberbezpieczeństwo, monitorowanie, procedury operacyjne i dokumentacja
- Odpowiedzialność inżynierska i zgodność z normami branżowymi

439

Dziękuję za uwagę

Wykład nr 10

Temat: Wybrane wzorce projektowe i ich implementacja

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

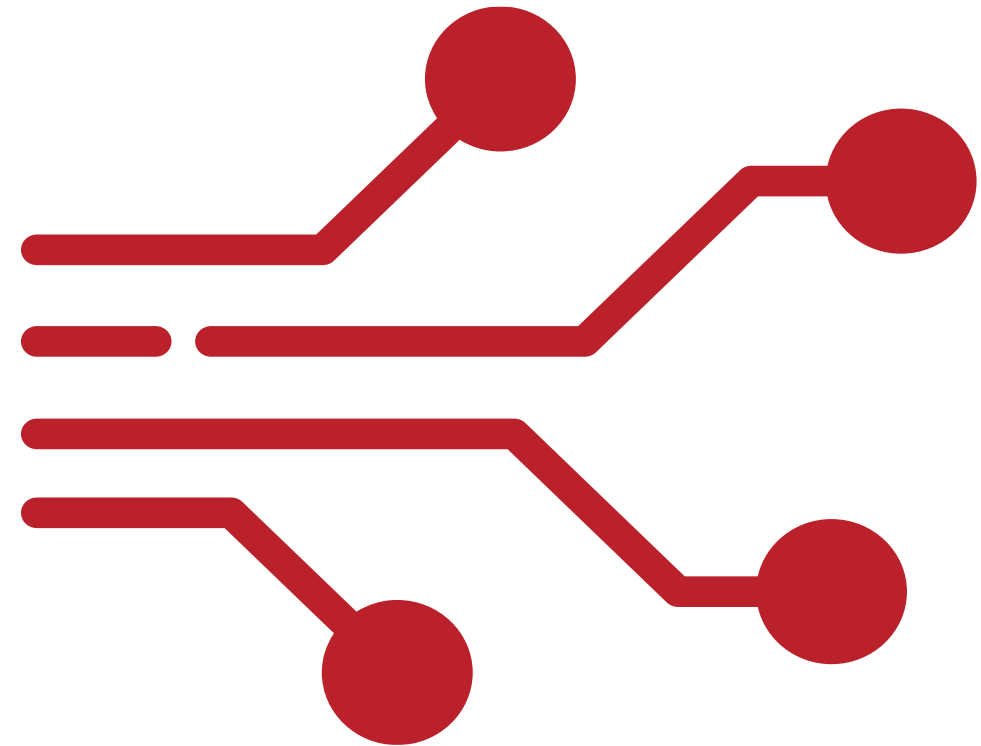
Plan prezentacji

- Wprowadzenie
- Podział
- Przykłady



442

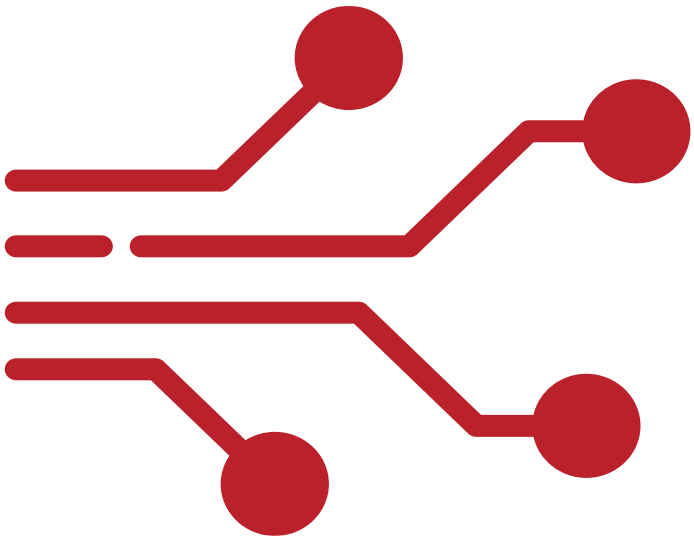
Wprowadzenie



443

Czym są wzorce projektowe?

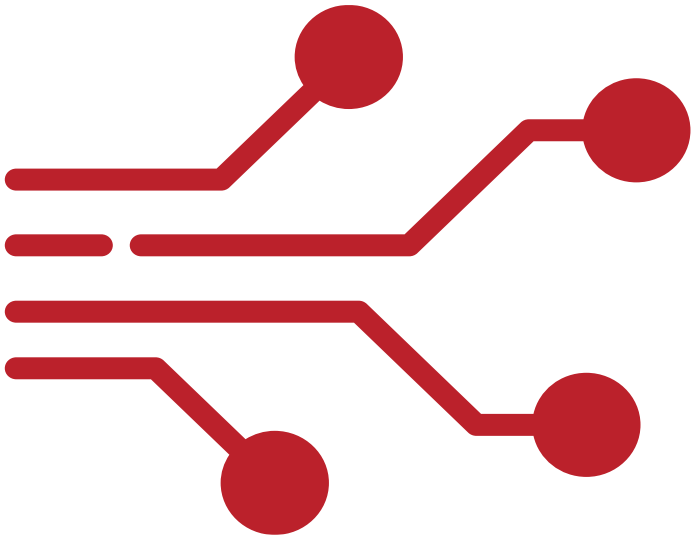
- Wzorce projektowe to sprawdzone rozwiązania typowych problemów projektowych
- Opisują sposób organizacji kodu i struktury systemu
- Nie są gotowym kodem, lecz ogólną koncepcją rozwiązania
- Pomagają tworzyć bardziej elastyczne i czytelne oprogramowanie
- Są szeroko stosowane w nowoczesnej inżynierii oprogramowania



444

Po co stosuje się wzorce projektowe

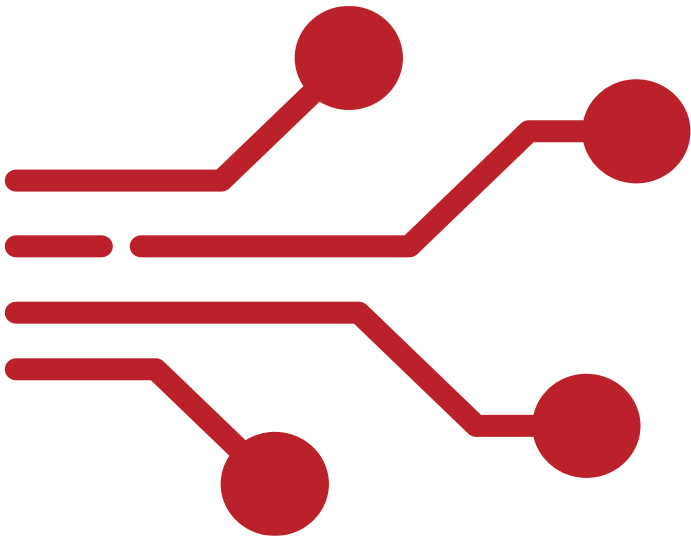
- Ułatwiają rozwiązywanie powtarzających się problemów projektowych
- Pomagają tworzyć bardziej czytelny i uporządkowany kod
- Zwiększają elastyczność i rozszerzalność systemu
- Ułatwiają współpracę programistów w zespole
- Wspierają utrzymanie i rozwój oprogramowania



445

Wzorzec projektowy a gotowe rozwiązanie

- Wzorzec projektowy nie jest gotowym fragmentem kodu
- Opisuje ogólną strukturę rozwiązania problemu
- Może być implementowany na wiele różnych sposobów
- Implementacja zależy od języka programowania i kontekstu systemu
- Ten sam wzorzec może mieć różne realizacje w różnych projektach



446

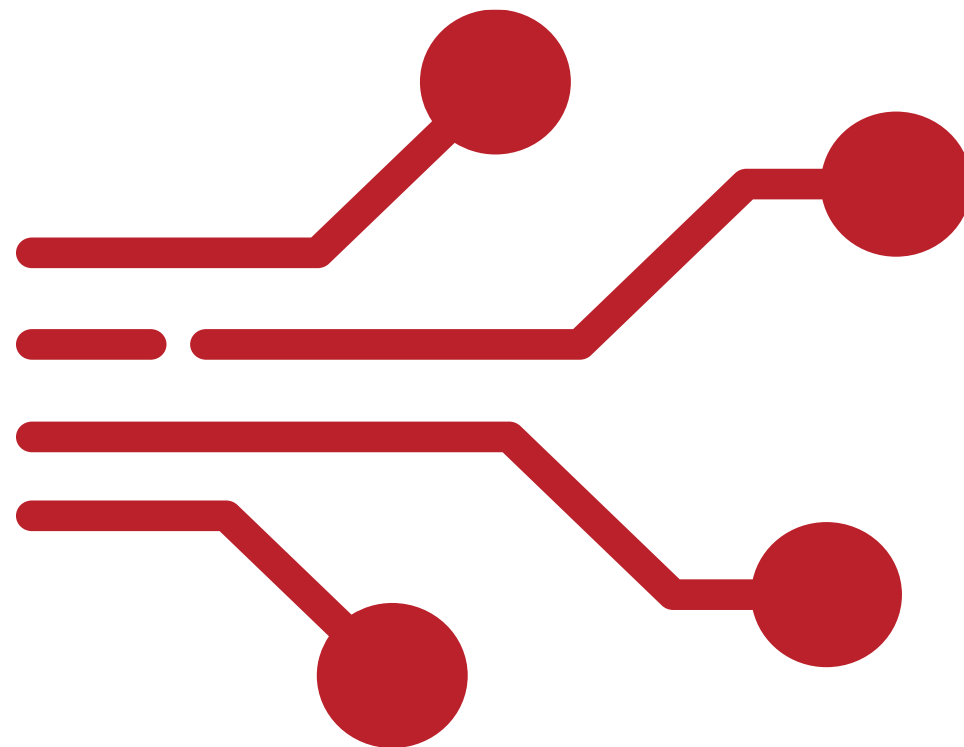
Gang of Four

- Erich Gamma - szwajcarski informatyk, współautor książki, pracował przy Eclipse IDE
- Richard Helm - brytyjski informatyk, współautor książki, specjalista w inżynierii oprogramowania
- Ralph Johnson - amerykański informatyk, specjalizuje się w obiektowych systemach i refaktoryzacji
- John Vlissides - amerykański informatyk, zajmował się wzorcami projektowymi i systemami OO

Książka: Design Patterns: Elements of Reusable Object-Oriented Software, 1994 - wprowadziła 23 klasyczne wzorce projektowe

447

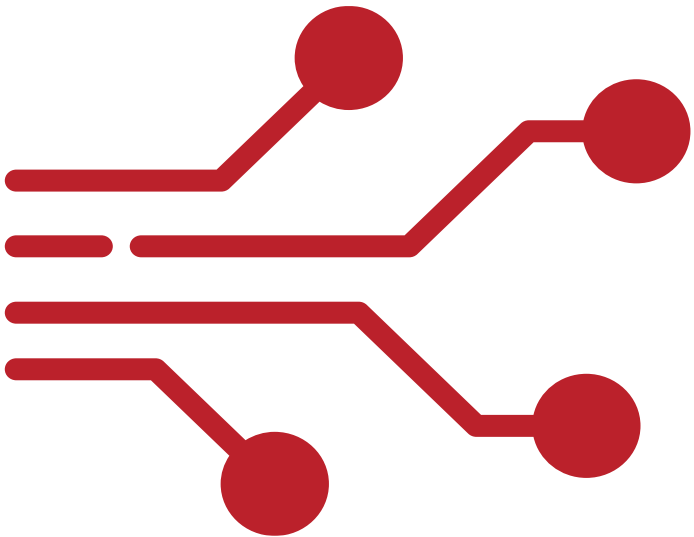
Podział



448

Kategorie wzorców projektowych

- Wzorce projektowe można podzielić na kilka głównych kategorii
- Najczęściej wyróżnia się trzy grupy wzorców
- Wzorce kreacyjne - dotyczą tworzenia obiektów
- Wzorce strukturalne - opisują organizację klas i obiektów
- Wzorce behawioralne - opisują sposób współpracy obiektów



Problem projektowy i kontekst użycia wzorca

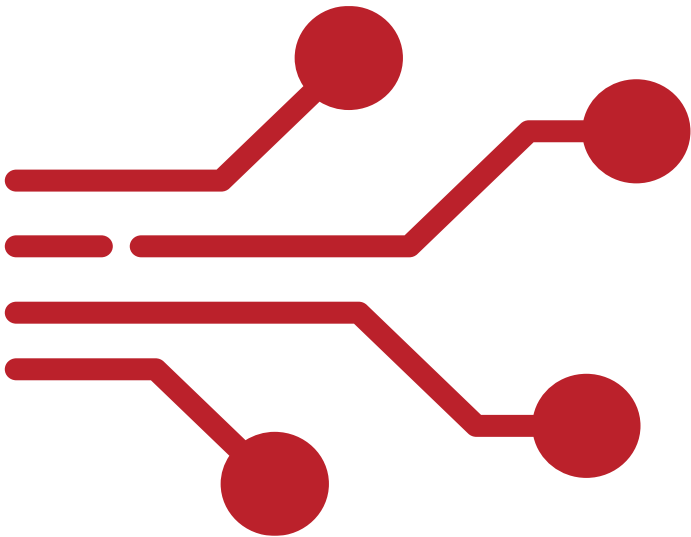
- Wzorce projektowe rozwiązują konkretne problemy projektowe
- Powinny być stosowane w określonym kontekście systemu
- Najpierw należy zrozumieć problem, a dopiero potem wybierać wzorzec
- Ten sam problem może mieć kilka możliwych rozwiązań
- Wybór wzorca zależy od architektury systemu



450

Zalety stosowania wzorców projektowych

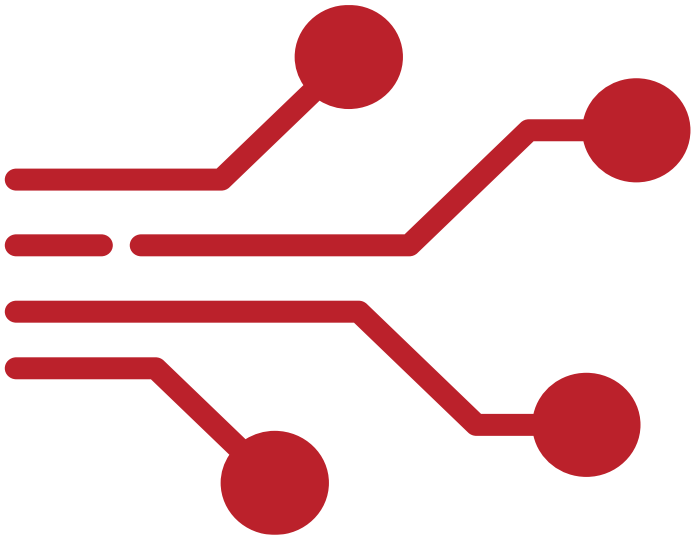
- Pomagają organizować strukturę systemu informatycznego
- Zmniejszają złożoność zależności między komponentami
- Ułatwiają rozbudowę i modyfikację systemu
- Zwiększają czytelność kodu dla innych programistów
- Pozwalają wykorzystywać sprawdzone rozwiązania projektowe



451

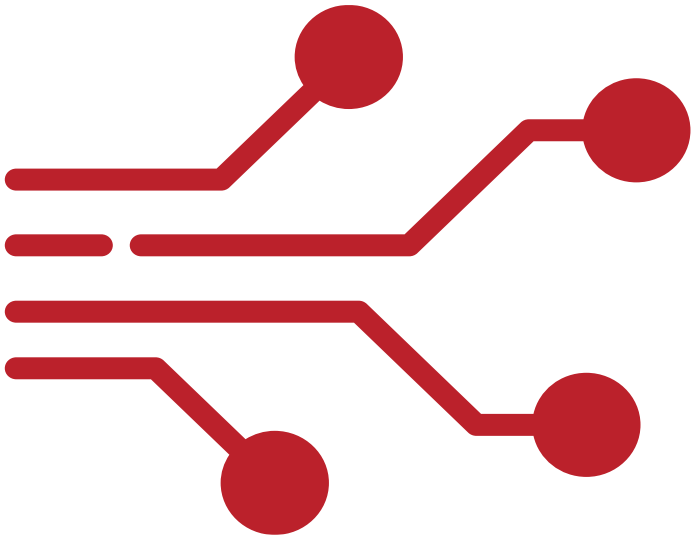
Ograniczenia stosowania wzorców

- Wzorce mogą zwiększać złożoność kodu
- Nie zawsze są potrzebne w prostych systemach
- Niewłaściwe użycie wzorca może utrudnić rozwój systemu
- Zbyt duża liczba abstrakcji utrudnia zrozumienie kodu
- Wzorce wymagają doświadczenia projektowego

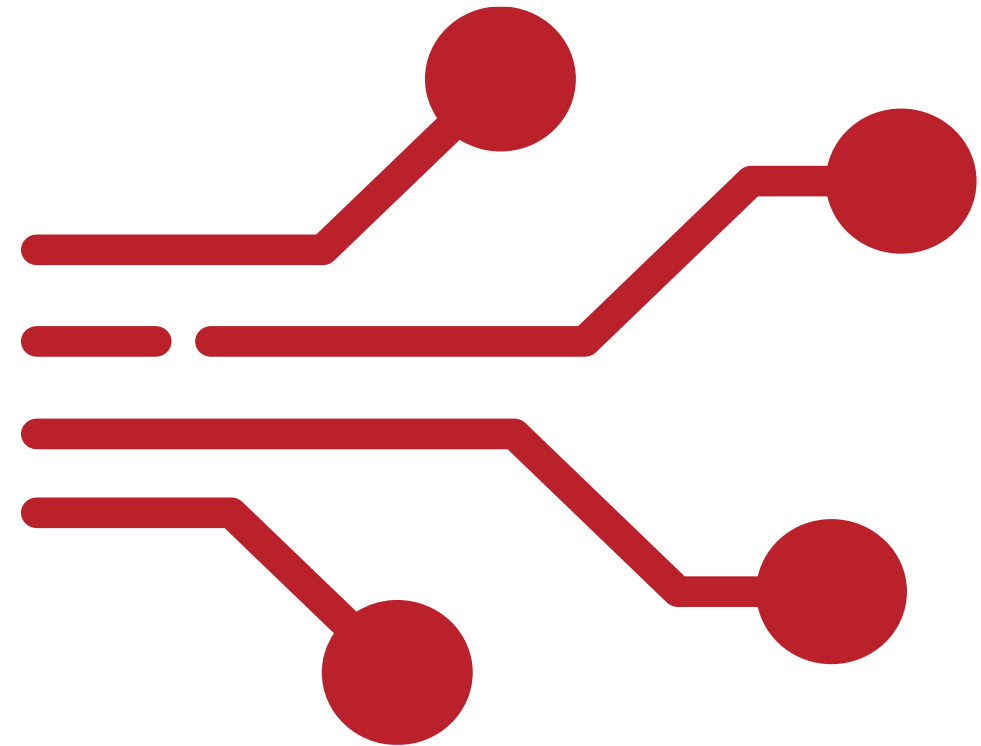


Kiedy nie używać wzorców projektowych

- Gdy problem projektowy jest bardzo prosty
- Gdy wzorzec wprowadza niepotrzebną złożoność
- Gdy rozwiązanie bez wzorca jest bardziej czytelne
- Gdy projekt jest bardzo mały lub jednorazowy
- Gdy zespół nie rozumie zastosowanego wzorca



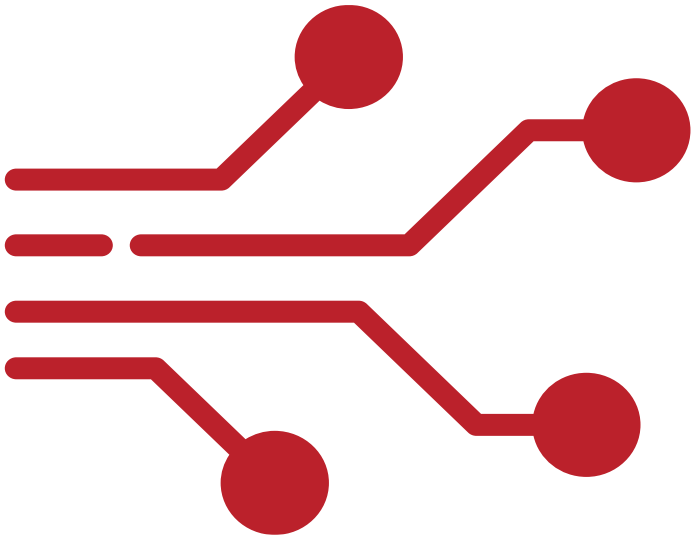
Przykłady



454

Wzorce kreacyjne

- Dotyczą sposobu tworzenia obiektów w systemie
- Oddzielają proces tworzenia obiektu od jego użycia
- Pozwalają kontrolować sposób inicjalizacji obiektów
- Ułatwiają tworzenie obiektów o złożonej strukturze
- Zwiększają elastyczność systemu



455

Factory Method

- Oddziela proces tworzenia obiektów od ich użycia
- Pozwala tworzyć obiekty bez wskazywania konkretnej klasy
- Umożliwia wybór typu obiektu w czasie działania programu
- Ułatwia rozbudowę systemu o nowe typy obiektów
- Jest często stosowany w systemach opartych na interfejsach

456

Factory Method

```
interface IAnimal
```

```
{
```

```
    void Speak();
```

```
}
```

```
class Dog : IAnimal
```

```
{
```

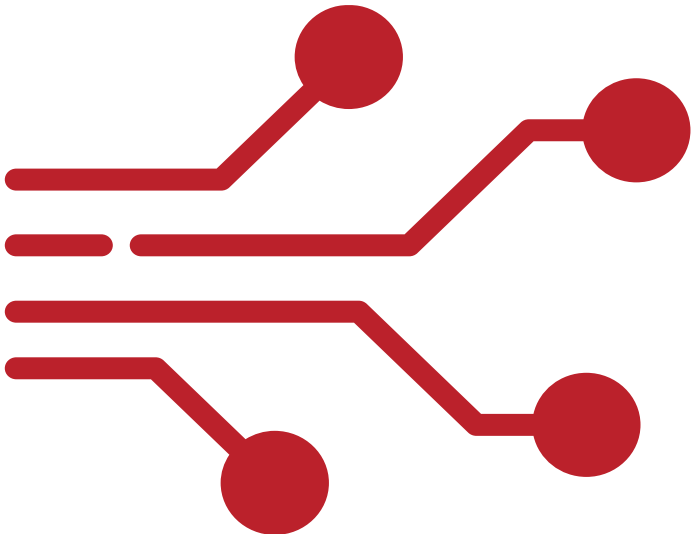
```
    public void Speak()
```

```
{
```

```
        Console.WriteLine("Bark");
```

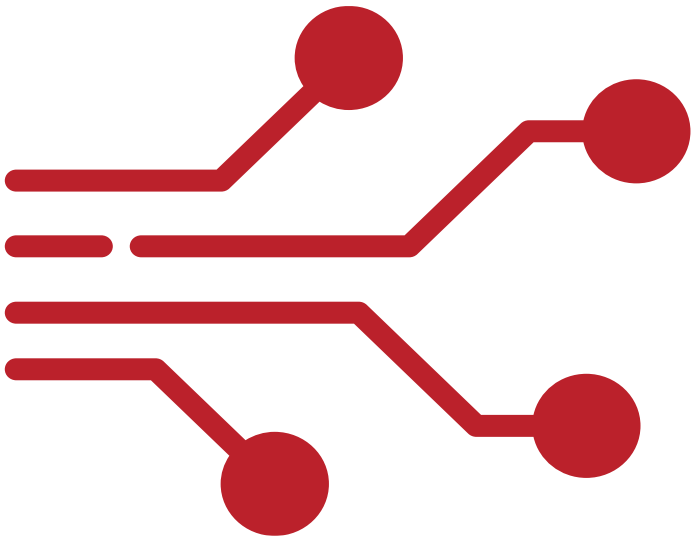
```
}
```

```
}
```



Factory Method

```
class Cat : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Meow");
    }
}
```



Factory Method

```
class AnimalFactory
```

```
{
```

```
    public static IAnimal Create(string type)
```

```
    {
```

```
        return type switch
```

```
        {
```

```
            "dog" => new Dog(),
```

```
            "cat" => new Cat(),
```

```
            _ => throw new ArgumentException("Unknown animal  
type")
```

```
        };
```

```
    }}
```

Builder

- Umożliwia tworzenie złożonych obiektów krok po kroku
- Oddziela proces konstruowania obiektu od jego reprezentacji
- Pozwala tworzyć różne warianty tego samego obiektu
- Ułatwia inicjalizację obiektów posiadających wiele parametrów
- Zwiększa czytelność kodu podczas tworzenia obiektów



460

Builder

```
class Computer
```

```
{
```

```
    public string CPU { get; set; }
```

```
    public int RAM { get; set; }
```

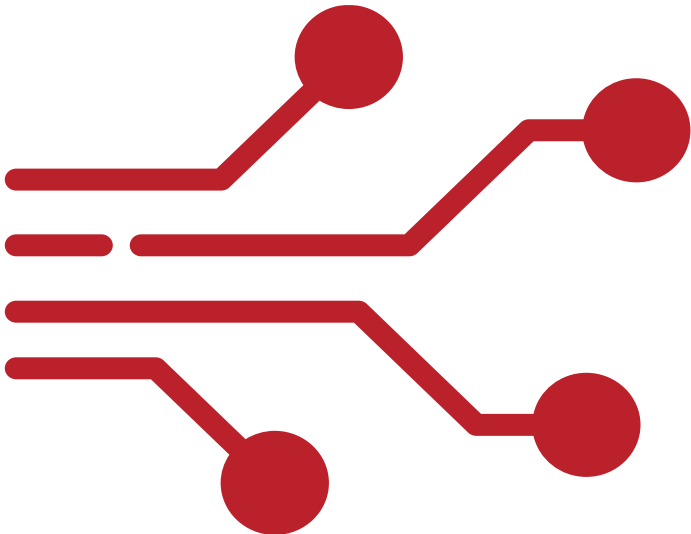
```
    public bool GPU { get; set; }
```

```
}
```

```
class ComputerBuilder
```

```
{
```

```
    private Computer computer = new Computer();
```



Builder

```
public ComputerBuilder SetCPU(string cpu)
{
    computer.CPU = cpu;
    return this;
}

public ComputerBuilder SetRAM(int ram)
{
    computer.RAM = ram;
    return this;
}
```

462

Builder

```
public ComputerBuilder SetGPU(bool gpu)
{
    computer.GPU = gpu;
    return this;
}

public Computer Build()
{
    return computer;
}
}
```

463

Wzorce strukturalne

- Dotyczą organizacji klas i obiektów w systemie
- Ułatwiają współpracę między komponentami
- Zmniejszają złożoność zależności między elementami systemu
- Pozwalają tworzyć bardziej przejrzystą i elastyczną architekturę
- Często stosowane przy integracji z istniejącymi komponentami



464

Adapter

- Umożliwia współpracę klas o niekompatybilnych interfejsach
- Działa jako pośrednik między dwoma systemami lub komponentami
- Pozwala używać istniejących klas bez modyfikowania ich kodu
- Ułatwia integrację nowych komponentów z systemem
- Jest często stosowany przy pracy z bibliotekami zewnętrznymi

465

Adapter

```
class OldPrinter
{
    public void PrintText(string text)
    {
        Console.WriteLine(text);
    }
}

interface IPrinter
{
    void Print(string message);
}
```

Adapter

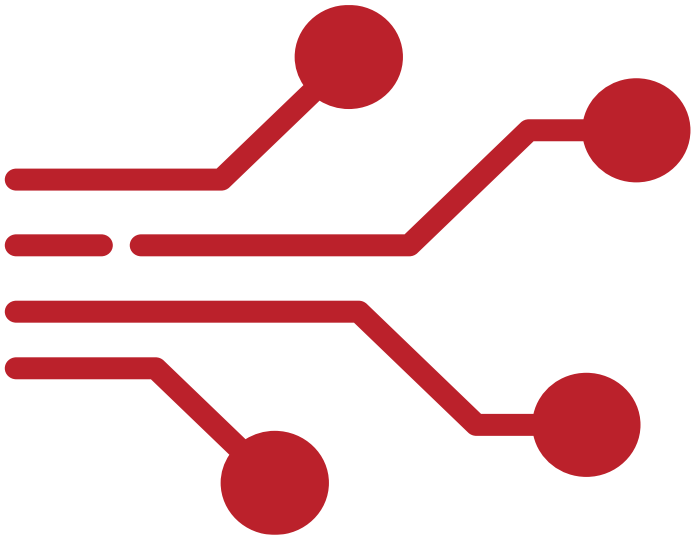
```
class PrinterAdapter : IPrinter
{
    private OldPrinter printer = new OldPrinter();

    public void Print(string message)
    {
        printer.PrintText(message);
    }
}
```

467

Facade

- Uproszcza dostęp do złożonego systemu lub grupy klas
- Tworzy jeden spójny interfejs dla wielu komponentów
- Ukrywa szczegóły implementacji przed klientem
- Ułatwia użycie systemu i ogranicza zależności
- Często stosowany w bibliotekach i systemach warstwowych



468

Facade

```
class Database
```

```
{
```

```
    public void Connect() => Console.WriteLine("Database  
connected");
```

```
}
```

```
class Logger
```

```
{
```

```
    public void Log(string message) =>  
Console.WriteLine("Log: " + message);
```

```
}
```

469

Facade

```
class SystemFacade
{
    private Database db = new Database();
    private Logger logger = new Logger();

    public void StartSystem()
    {
        db.Connect();
        logger.Log("System started");
    }
}
```

470

Wzorce behawioralne

- Dotyczą sposobu współpracy i komunikacji między obiektami
- Pozwalają rozdzielić odpowiedzialności między komponentami
- Ułatwiają zmiany w zachowaniu systemu bez modyfikacji całej struktury
- Często stosowane przy implementacji algorytmów i reakcji na zdarzenia
- Zwiększają elastyczność i rozszerzalność systemu

471

Strategy

- Pozwala definiować rodzinę algorytmów i wymieniać je w czasie działania programu
- Oddziela algorytm od klienta, który go używa
- Ułatwia dodawanie nowych sposobów działania bez modyfikowania istniejącego kodu
- Zwiększa elastyczność i rozszerzalność systemu
- Często stosowany przy wyborze metod obliczeniowych, walidacji czy przetwarzania danych

472

Strategy

```
interface IPaymentStrategy
```

```
{
```

```
    void Pay(int amount);
```

```
}
```

```
class CreditCardPayment : IPaymentStrategy
```

```
{
```

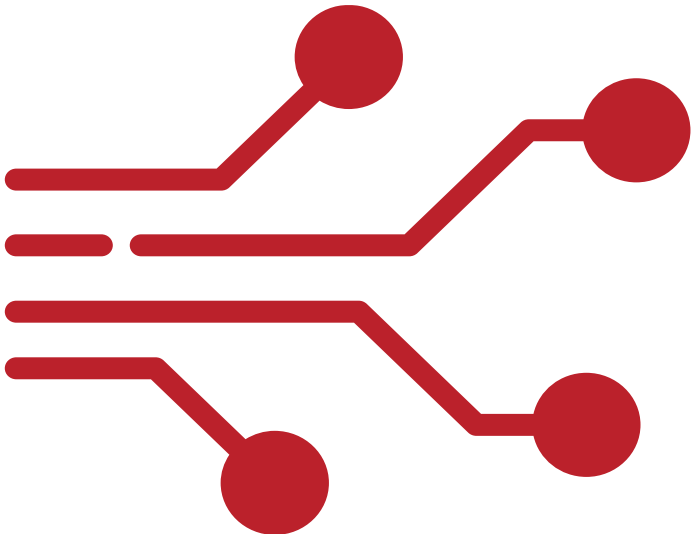
```
    public void Pay(int amount)
```

```
    {
```

```
        Console.WriteLine($"Paid {amount} using credit  
card.");
```

```
    }}
```

Strategy



```
class PayPalPayment : IPaymentStrategy
{
    public void Pay(int amount)
    {
        Console.WriteLine($"Paid {amount} using PayPal.");
    }
}
```

474

Strategy

```
class PaymentProcessor
{
    private IPaymentStrategy strategy;

    public PaymentProcessor(IPaymentStrategy strategy)
    {
        this.strategy = strategy;
    }
    public void ProcessPayment(int amount)
    {
        strategy.Pay(amount);
    }
}
```

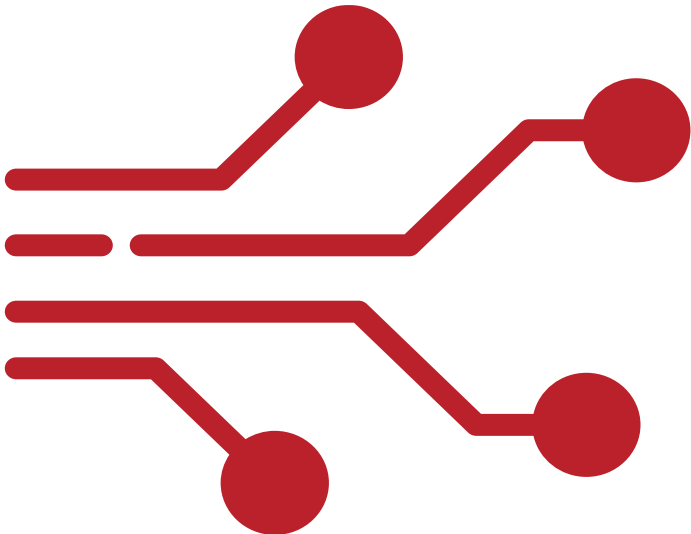
475

Observer

- Umożliwia powiadamianie wielu obiektów o zmianach w innym obiekcie
- Oddziela źródło zdarzeń od odbiorców
- Pozwala dynamicznie dodawać lub usuwać obserwatorów w czasie działania programu
- Ułatwia implementację mechanizmu powiadomień i subskrypcji
- Często stosowany w GUI, systemach zdarzeniowych i komunikacji między komponentami

476

Observer
using System;



```
class NewsAgency
{
    public event Action<string> NewsPublished;

    public void Publish(string news)
    {
        NewsPublished?.Invoke(news);
    }
}
```

477

Observer

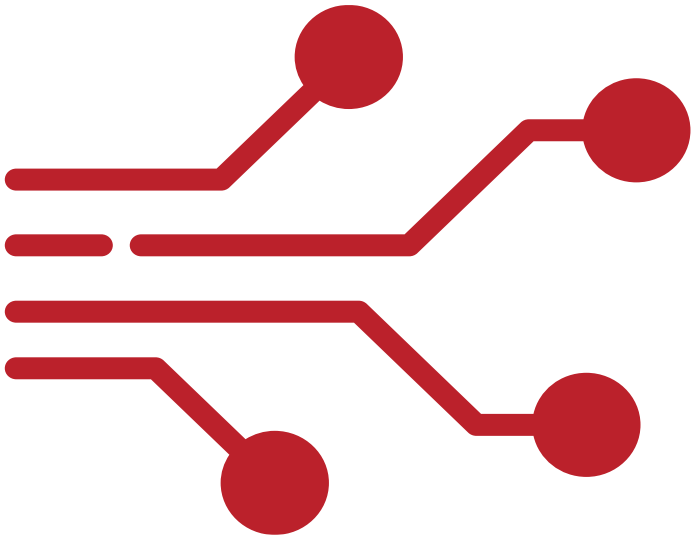
```
class Subscriber
```

```
{  
    public string Name { get; }  
    public Subscriber(string name) => Name = name;  
  
    public void ReceiveNews(string news)  
    {  
        Console.WriteLine($"{Name} received news: {news}");  
    }  
}
```

478

Wzorce w praktyce: jak rozpoznać potrzebę ich użycia?

- Najpierw identyfikujemy powtarzające się problemy projektowe
- Zastanawiamy się, czy istnieje sprawdzone rozwiązanie (wzorzec)
- Analizujemy zależności między klasami i obiektami
- Sprawdzamy, czy wzorzec zwiększy elastyczność systemu
- Rozważamy wpływ wzorca na czytelność i utrzymanie kodu



479

Refaktoryzacja kodu do wzorca

- Identyfikujemy fragmenty kodu powtarzające się lub trudne w utrzymaniu
- Wybieramy odpowiedni wzorzec projektowy
- Przekształcamy kod tak, aby spełniał strukturę wzorca
- Testujemy, aby zachować dotychczasową funkcjonalność
- Dokumentujemy zmiany i sposób użycia wzorca w systemie



480

Wzorce a testowalność kodu

- Ułatwiają izolowanie komponentów w testach jednostkowych
- Zmniejszają zależności między klasami
- Pozwalają łatwo podmieniać implementacje (mocki, stuby)
- Ułatwiają wprowadzanie testów dla różnych wariantów działania
- Zwiększają przewidywalność i niezawodność testów

481

Wzorce a utrzymanie systemu

- Ułatwiają wprowadzanie nowych funkcjonalności
- Zmniejszają o wprowadzania błędów przy zmianach
- Poprawiają czytelność i spójność kodu
- Ułatwiają rozbudowę i integrację z innymi modułami
- Umożliwiają stosowanie sprawdzonych, powtarzalnych rozwiązań



Typowe błędy w stosowaniu wzorców

- Nadmierne stosowanie wzorców bez rzeczywistej potrzeby
- Tworzenie zbyt wielu abstrakcji, które utrudniają czytelność
- Stosowanie wzorców w prostych systemach, gdzie są zbędne
- Nieodpowiedni wybór wzorca do konkretnego problemu
- Brak dokumentacji i wyjaśnienia zastosowanego wzorca

483

Antywzorce (Anti-Patterns)

Powtarzalne błędy projektowe, które prowadzą do problemów w kodzie i architekturze. Mogą wynikać z:

- Nieprzemyślanego stosowania wzorców projektowych
- Zbyt dużej złożoności lub nadmiarowej abstrakcji
- Braku modularności i silnych zależności między klasami
- Przykłady typowych antywzorców:
 - God Object - jedna klasa robi wszystko, nadmiar odpowiedzialności
 - Spaghetti Code - zbyt powiązany i nieczytelny kod
 - Poltergeist - klasy tymczasowe lub „puste”, przekazujące dane bez logiki
 - Golden Hammer - nadmierne stosowanie jednego rozwiązania do każdego problemu
 - Lava Flow - nieusuwany, martwy kod, który pozostaje w systemie

484

Podsumowanie wykładu

- Wzorce projektowe to sprawdzone rozwiązania typowych problemów projektowych
- Dzielią się na grupy: kreacyjne, strukturalne i behawioralne
- Stosowane świadomie zwiększają elastyczność, czytelność i testowalność kodu
- Pomagają w utrzymaniu systemu i rozbudowie funkcjonalności
- Należy unikać nadmiernego stosowania i błędów w doborze wzorców

485

Dziękuję za uwagę

Wykład nr 11 **Temat: Zarządzanie projektem** **informatycznym. Cechy** **projektu. Wybrane aspekty** **zarządzania projektem.** **Wybrane narzędzia do** **zarządzania projektami**

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

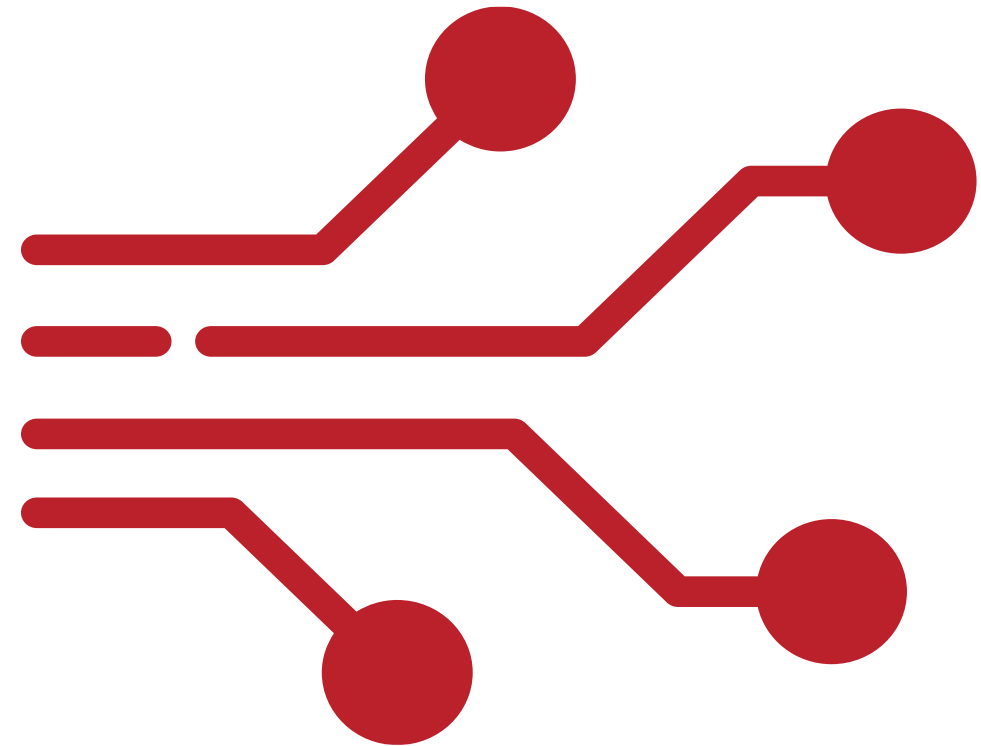
Plan prezentacji

- Projekt i jego cechy
- Wybrane aspekty zarządzania



488

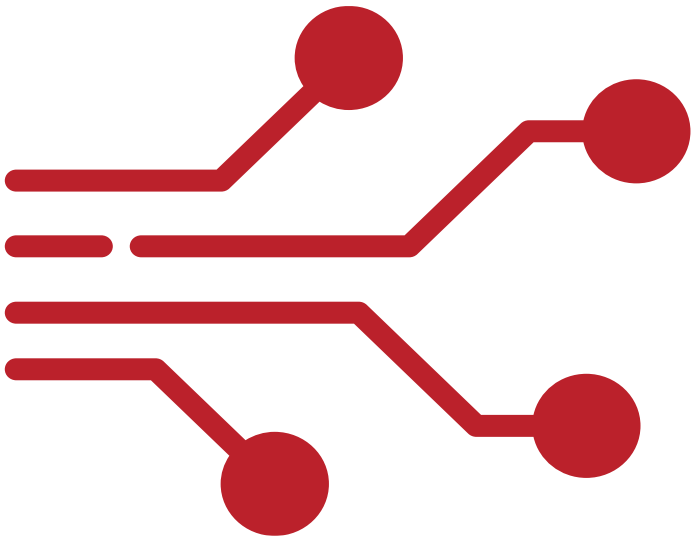
Wprowadzenie



489

Projekt informatyczny

- Projekt to tymczasowe przedsięwzięcie mające określony cel
- Prowadzi do stworzenia produktu, systemu lub usługi informatycznej
- Ma jasno określony początek i koniec
- Realizowany jest przez zespół projektowy
- Wymaga planowania, koordynacji i kontroli działań



490

Cechy projektu informatycznego

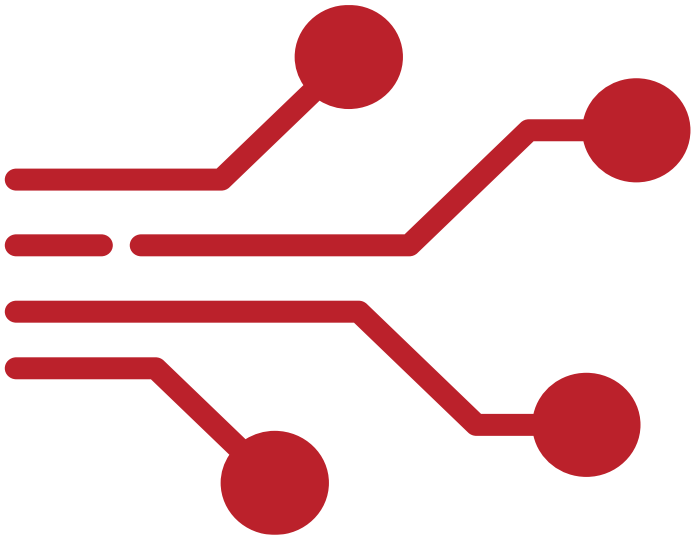
- Ma jasno określony cel
- Posiada określony czas trwania
- Wykorzystuje określone zasoby
- Prowadzi do powstania unikalnego produktu lub systemu
- Wiąże się z niepewnością i ryzykiem



491

Projekt a działalność operacyjna

- Projekty mają charakter tymczasowy
- Działalność operacyjna ma charakter ciągły
- Projekt prowadzi do stworzenia nowego rozwiązania
- Działalność operacyjna polega na wykonywaniu powtarzalnych działań
- Projekty wymagają zarządzania zmianą i ryzykiem



492

Dlaczego zarządzanie projektem jest ważne

- Umożliwia realizację projektu w określonym czasie
- Pozwala kontrolować koszty projektu
- Pomaga koordynować pracę zespołu
- Ułatwia reagowanie na zmiany i problemy
- Zwiększa prawdopodobieństwo sukcesu projektu



Specyfika projektów informatycznych

- Wysoki poziom złożoności systemów
- Częste zmiany wymagań w trakcie projektu
- Duża zależność od technologii
- Wysoki poziom a projektowego
- Trudność w dokładnym szacowaniu czasu i kosztów



494

Cele projektu i kryteria sukcesu

- Projekt powinien posiadać jasno określony cel
- Cele projektu powinny być mierzalne
- Cele powinny być zgodne z potrzebami organizacji lub klienta
- Kryteria sukcesu określają, kiedy projekt uznaje się za zakończony powodzeniem
- Cele projektu powinny być znane wszystkim członkom zespołu

495

Zakres projektu

- Zakres projektu określa, co ma zostać wykonane w projekcie
- Definiuje funkcjonalności i elementy systemu
- Określa granice projektu
- Pomaga uniknąć niekontrolowanego rozszerzania projektu
- Stanowi podstawę planowania prac



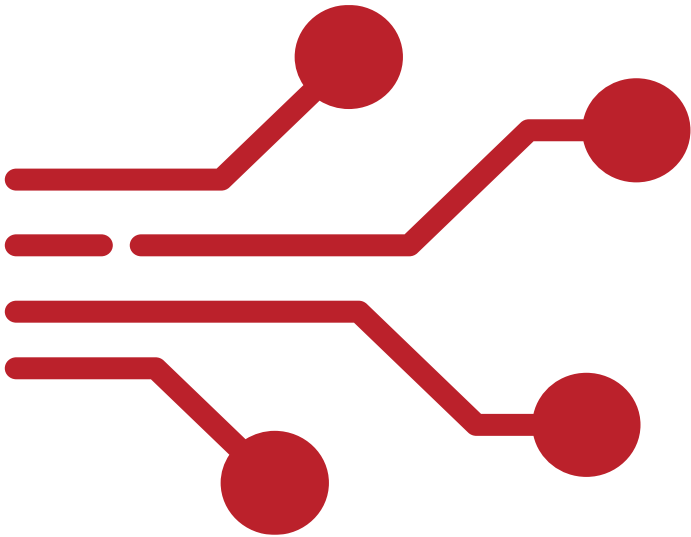
Interesariusze projektu

- Interesariusze to osoby lub organizacje związane z projektem
- Mogą wpływać na przebieg projektu lub jego rezultaty
- Mogą być użytkownikami systemu lub osobami decyzyjnymi
- Mają określone oczekiwania wobec projektu
- Wymagają odpowiedniej komunikacji i zarządzania oczekiwaniami

497

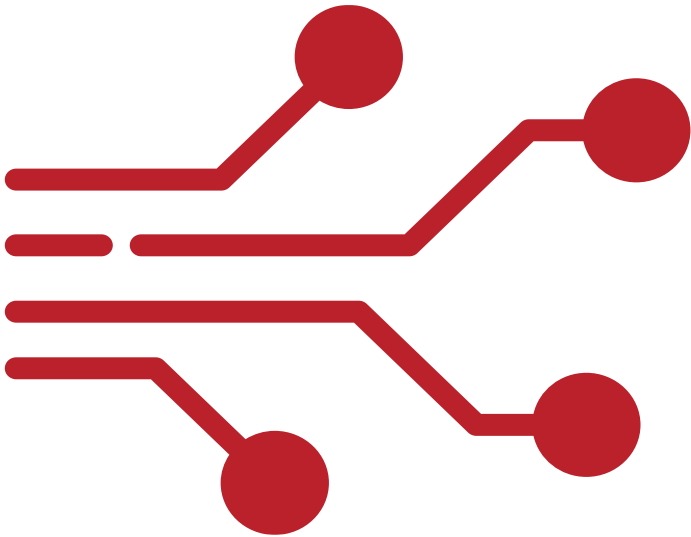
Ograniczenia projektowe: zakres, czas, koszt, jakość

- Projekty realizowane są w określonych ograniczeniach
- Najważniejsze ograniczenia to zakres, czas i koszt
- Często uwzględnia się również jakość produktu
- Zmiana jednego elementu wpływa na pozostałe
- Zarządzanie projektem polega na utrzymaniu równowagi między tymi elementami



Równowaga między czasem, kosztem i zakresem

- Czas, koszt i zakres tworzą tzw. trójkąt projektowy
- Zmiana jednego elementu wpływa na pozostałe
- Zwiększenie zakresu zwykle wydłuża czas lub zwiększa koszt
- Skrócenie czasu często wymaga większych zasobów
- Kierownik projektu musi utrzymywać równowagę między tymi elementami



Fazy projektu informatycznego

- Inicjowanie projektu
- Planowanie projektu
- Realizacja projektu
- Monitorowanie i kontrola
- Zamknięcie projektu



500

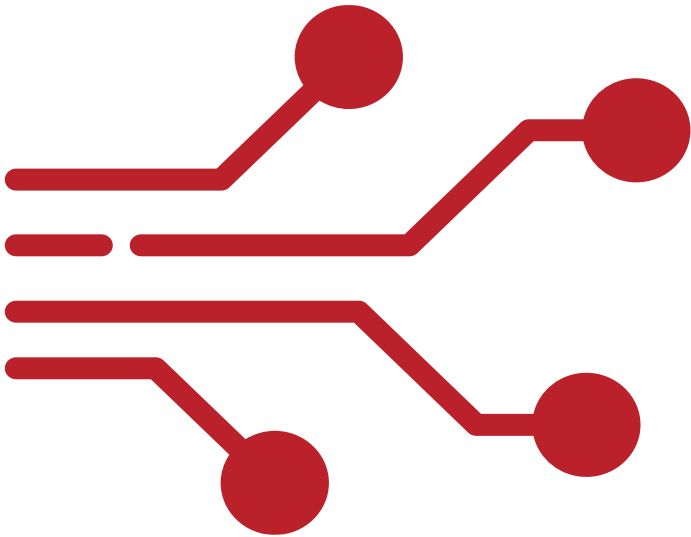
Inicjowanie projektu

- Identyfikacja potrzeby lub problemu biznesowego
- Określenie celu projektu
- Wstępna analiza wykonalności projektu
- Identyfikacja głównych interesariuszy
- Decyzja o rozpoczęciu projektu



Planowanie projektu

- Określenie szczegółowego zakresu projektu
- Podział projektu na zadania
- Przygotowanie harmonogramu prac
- Określenie zasobów potrzebnych do realizacji projektu
- Oszacowanie kosztów projektu



502

Realizacja projektu

- Wykonywanie zaplanowanych zadań projektowych
- Tworzenie oprogramowania lub systemu informatycznego
- Współpraca członków zespołu projektowego
- Rozwiązywanie problemów technicznych i organizacyjnych
- Monitorowanie postępów prac



Monitorowanie i kontrola projektu

- Śledzenie postępów realizacji zadań
- Porównywanie postępów z harmonogramem i budżetem
- Identyfikacja odchyleń i problemów
- Podejmowanie działań korygujących
- Raportowanie wyników interesariuszom



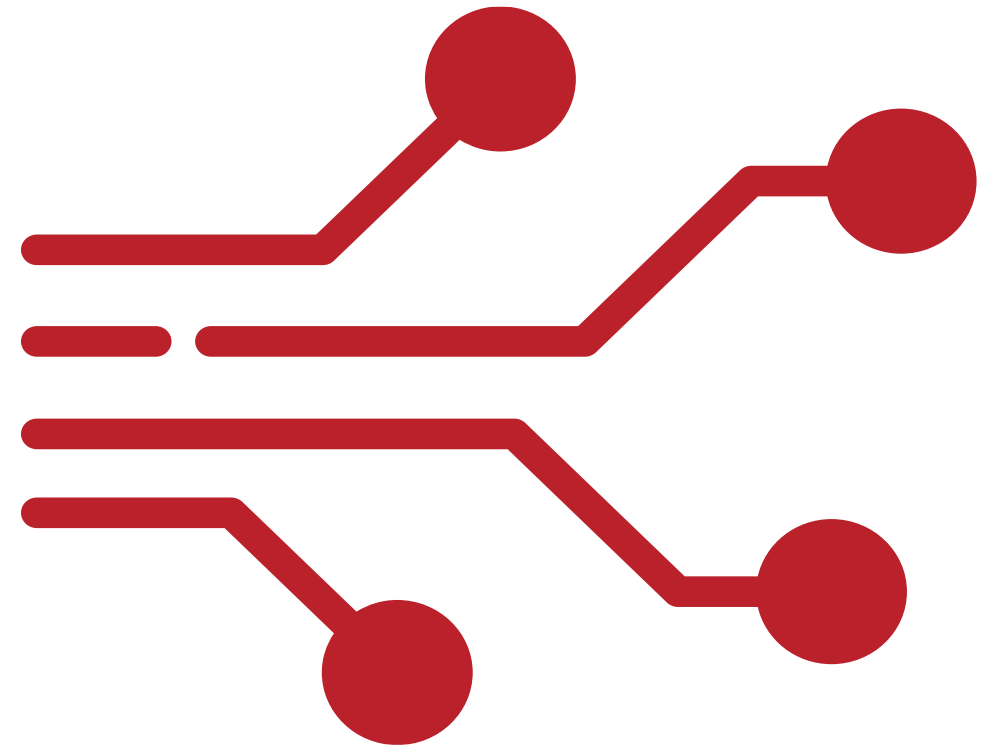
Zamknięcie projektu

- Przekazanie gotowego systemu lub produktu użytkownikowi
- Podsumowanie wykonanych zadań i osiągniętych celów
- Ocena jakości wykonania projektu
- Dokumentowanie wniosków i doświadczeń
- Formalne zakończenie projektu i zwolnienie zasobów



505

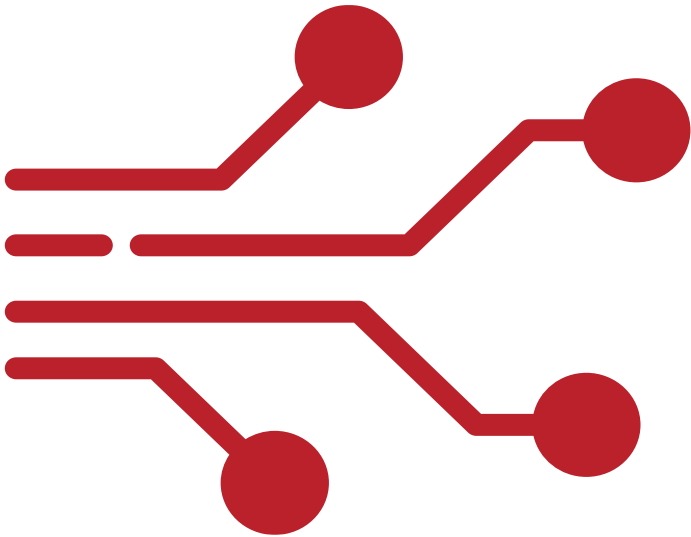
Wybrane aspekty zarządzania



506

Identyfikacja zadań projektowych

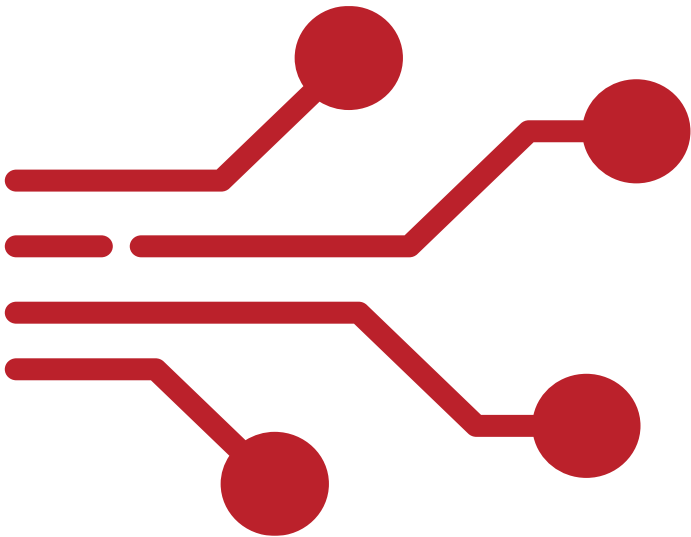
- Podział projektu na mniejsze, konkretne zadania
- Określenie odpowiedzialności za każde zadanie
- Ustalenie priorytetów zadań
- Powiązanie zadań z kamieniami milowymi
- Podstawa do harmonogramu i szacowania czasu



507

Struktura podziału pracy (WBS)

- WBS (Work Breakdown Structure) to hierarchiczny podział projektu
- Dzieli projekt na zadania, podzadania i elementy pracy
- Ułatwia planowanie, harmonogramowanie i przypisywanie zasobów
- Pozwala monitorować postępy i kontrolować wykonanie
- Stanowi podstawę szacowania czasu i kosztów



508

WBS (Work Breakdown Structure)

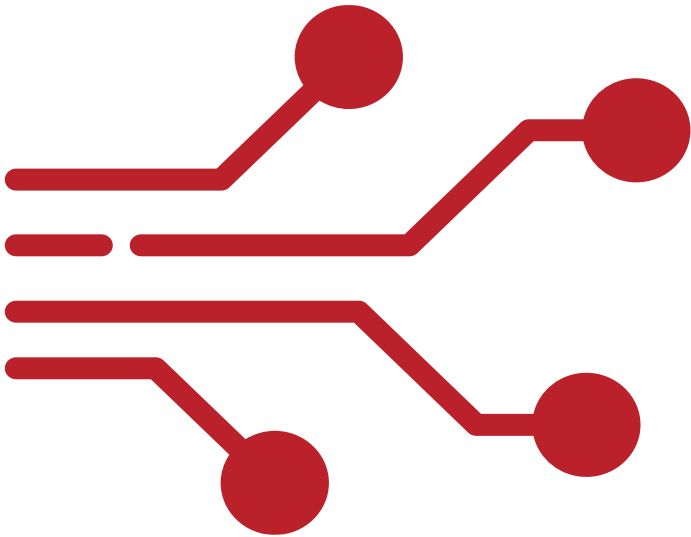
- Hierarchiczny podział projektu na zadania i podzadania
- Każde zadanie ma określony czas, zasoby i odpowiedzialność
- Ułatwia planowanie, szacowanie kosztów i harmonogramowanie
- Pomaga monitorować postęp realizacji projektu
- Jest podstawą do tworzenia diagramów Gantta i harmonogramów



509

PBS (Product Breakdown Structure)

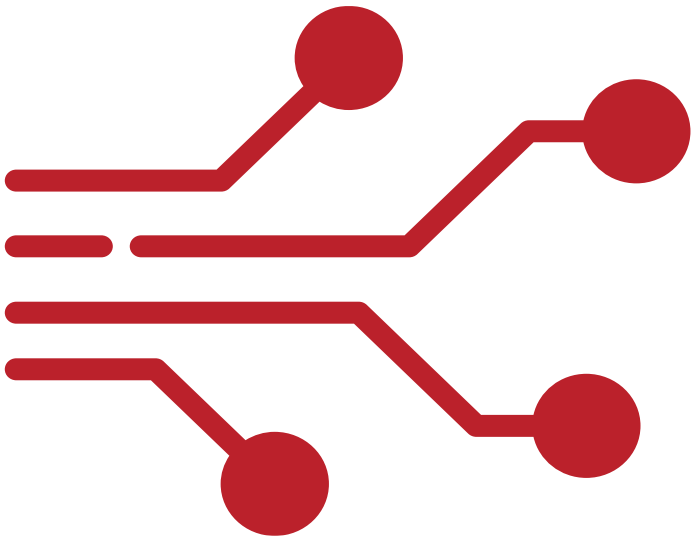
- Hierarchiczny podział produktu na komponenty i podkomponenty
- Skupia się na produkcie końcowym, a nie na zadaniach
- Ułatwia identyfikację elementów do dostarczenia
- Pomaga powiązać komponenty produktu z wymaganiami i testami
- Podstawa do planowania WBS i szacowania kosztów



510

Ścieżka krytyczna (Critical Path)

- Najdłuższa sekwencja zależnych zadań w projekcie
- Określa minimalny czas potrzebny do realizacji projektu
- Zadania na ścieżce krytycznej nie mogą być opóźnione bez wpływu na cały projekt
- Pozwala identyfikować zadania krytyczne i priorytety w harmonogramie
- Wykorzystywana w diagramach sieciowych i Gantta



511

Kamienie milowe projektu

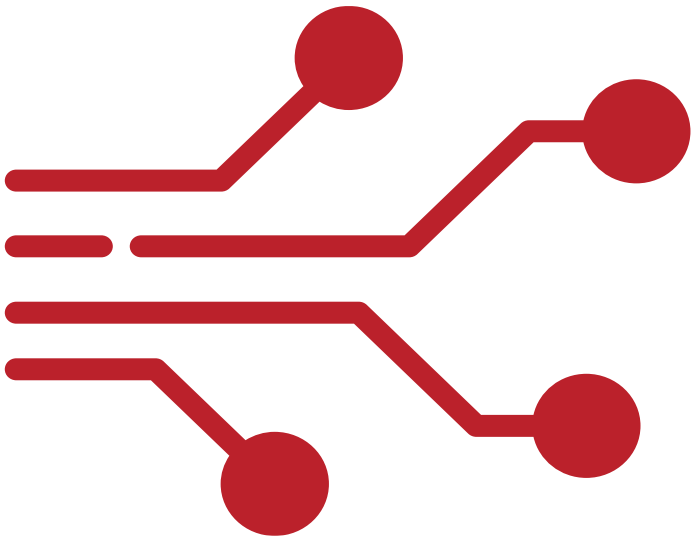
- Kluczowe punkty w harmonogramie projektu
- Wyznaczają zakończenie istotnych etapów
- Pozwalają śledzić postępy i osiągnięcie celów
- Pomagają w raportowaniu interesariuszom
- Ułatwiają zarządzanie ryzykiem i zmianami



512

Szacowanie czasu trwania zadań

- Określenie, ile czasu potrzeba na wykonanie każdego zadania
- Uwzględnienie dostępnych zasobów i doświadczenia zespołu
- Wykorzystanie metod szacowania: eksperckiej, analogicznej, bottom-up
- Uwzględnienie marginesu na a i nieprzewidziane opóźnienia
- Podstawa do stworzenia harmonogramu projektu



Szacowanie kosztu projektu

- Określenie wszystkich kosztów związanych z projektem
- Uwzględnienie kosztów zasobów ludzkich, sprzętu i oprogramowania
- Włączenie kosztów pośrednich, np. energii, licencji, szkoleń
- Szacowanie kosztów a i rezerw na nieprzewidziane wydatki
- Podstawa do przygotowania budżetu projektu

514

Zasoby projektu

- Ludzkie: programiści, analitycy, testerzy, administratorzy
- Sprzętowe: serwery, komputery, urządzenia sieciowe
- Finansowe: budżet projektu, rezerwy na ryzyka
- Technologiczne: oprogramowanie, licencje, narzędzia deweloperskie
- Czasowe: dostępność zespołu i harmonogram projektu



515

Role w zespole projektowym

- Kierownik projektu zarządza całym projektem
- Programiści tworzą kod i implementują funkcjonalności
- Analitycy systemowi określają wymagania i projektują rozwiązania
- Testerzy weryfikują jakość i poprawność systemu
- Administratorzy i DevOps utrzymanie środowisk i wdrożenia



516

Komunikacja w zespole projektowym

- Jasne określanie zadań i oczekiwań
- Regularne spotkania i raportowanie postępów
- Wykorzystywanie narzędzi do komunikacji (Slack, Teams, Jira)
- Ustalanie priorytetów i terminów realizacji
- Ułatwienie wymiany informacji między rolami



517

Metoda SMART w przekazywaniu zadań

- Specific (konkretne) - jasno określone zadanie
- Measurable (mierzalne) - określony sposób weryfikacji wykonania
- Achievable (osiągalne) - realistyczne w danych warunkach
- Relevant (istotne) - powiązane z celami projektu
- Time-bound (określone w czasie) - z jasno określonym terminem

518

Feedback i monitorowanie wykonania zadań

- Regularne sprawdzanie postępów prac
- Udzielanie konstruktywnego feedbacku
- Wyjaśnianie wątpliwości i problemów
- Dostosowywanie priorytetów i terminów w razie potrzeby
- Motywowanie zespołu do efektywnej realizacji zadań



Zarządzanie ryzykiem projektu

- Identyfikacja potencjalnych zagrożeń i problemów
- Ocena prawdopodobieństwa wystąpienia i wpływu ryzyka
- Opracowanie planów reakcji na ryzyko
- Monitorowanie ryzyka w trakcie realizacji projektu
- Dokumentowanie ryzyk i działań korygujących



Zarządzanie zmianą w projekcie

- Zmiany mogą dotyczyć zakresu, czasu, kosztu lub wymagań
- Identyfikacja źródła zmiany i jej wpływu na projekt
- Ocena, czy zmiana jest niezbędna i opłacalna
- Wdrożenie formalnego procesu zatwierdzania zmian
- Dokumentowanie zmian i komunikacja z interesariuszami

521

Zarządzanie konfliktem w zespole

- Konflikty są naturalną częścią pracy zespołowej
- Mogą wynikać z różnic w celach, priorytetach lub stylach pracy
- Wczesne wykrycie konfliktu ułatwia jego rozwiązanie
- Stosowanie mediacji i negocjacji w celu osiągnięcia kompromisu
- Konflikty dobrze zarządzane mogą poprawić komunikację i innowacyjność

522

Szacowanie czasu całego projektu

- Sumowanie czasu wszystkich zadań i podzadań
- Uwzględnienie zależności między zadaniami
- Wykorzystanie buforów czasowych na ryzyka i nieprzewidziane opóźnienia
- Metody: analiza historyczna, Delphi, PERT
- Podstawa do harmonogramu i zarządzania terminami



523

Szacowanie kosztu całego projektu

- Sumowanie kosztów wszystkich zadań, zasobów i działań pośrednich
- Uwzględnienie kosztów ryzyk i nieprzewidzianych wydatków
- Kategorie kosztów: zasoby ludzkie, sprzęt, oprogramowanie, licencje, szkolenia
- Metody szacowania: analogiczne, parametryczne, bottom-up
- Podstawa do przygotowania budżetu projektu i kontroli wydatków

524

Metody szacowania wielkości projektu

- Szacowanie wielkości projektu pozwala określić czas, koszty i zasoby potrzebne do realizacji
- Podstawowe podejścia:
 - Analogiczne - porównanie z wcześniejszymi podobnymi projektami
 - Parametryczne - wykorzystanie wzorów i wskaźników (np. koszt na funkcję lub linia kodu)
 - Bottom-up (oddolne) - sumowanie kosztów i czasu wszystkich zadań i podzadań
 - Expert judgment/Delphi - ocena ekspertów w zespole projektowym
- Wybór metody zależy od dostępnych danych i doświadczenia zespołu

Metoda analogiczna

- Szacowanie w oparciu o podobne, zakończone projekty
- Zalety: szybka, łatwa do wdrożenia
- Wady: mniej dokładna, wymaga danych historycznych

Metoda parametryczna

- Wykorzystuje formuły i wskaźniki (np. koszt na funkcję, linie kodu, story points)
- Zalety: bardziej obiektywna, łatwa do aktualizacji
- Wady: wymaga dobrego dobrania parametrów i danych wejściowych

526

Metoda bottom-up (oddolna)

- Szacowanie czasu i kosztu dla każdego zadania w WBS
- Sumowanie wszystkich zadań daje całkowity czas i koszt projektu
- Zalety: bardzo dokładna, uwzględnia szczegóły projektu
- Wady: czasochłonna, wymaga pełnego planu zadań

Expert Judgment/Delphi

- Wykorzystanie doświadczenia ekspertów w szacowaniu wielkości projektu
- Konsensus ekspertów minimalizuje błędy indywidualne
- Zalety: szybkie, uwzględnia niestandardowe ryzyka
- Wady: subiektywne, wymaga ekspertów z doświadczeniem

527

Przydział zasobów i harmonogramowanie

- Przypisanie odpowiednich osób do zadań projektowych
- Uwzględnienie dostępności i kompetencji członków zespołu
- Przydział sprzętu, narzędzi i oprogramowania do zadań
- Harmonogramowanie zadań z uwzględnieniem zależności i priorytetów
- Monitorowanie wykorzystania zasobów i dostosowywanie planu

528

Monitorowanie postępu projektu

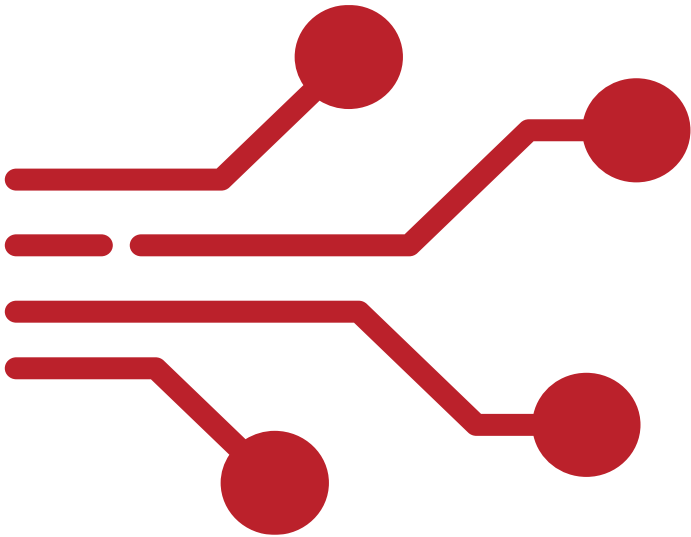
- Wykres Gantta - wizualizacja harmonogramu zadań
- Metoda kamieni milowych - śledzenie kluczowych etapów
- Metody iteracyjne - Agile, Scrum, monitorowanie sprintów
- Kontrola czasu, kosztów i jakości realizacji zadań
- Raportowanie statusu do interesariuszy



529

Diagram Gantt - definicja i cel

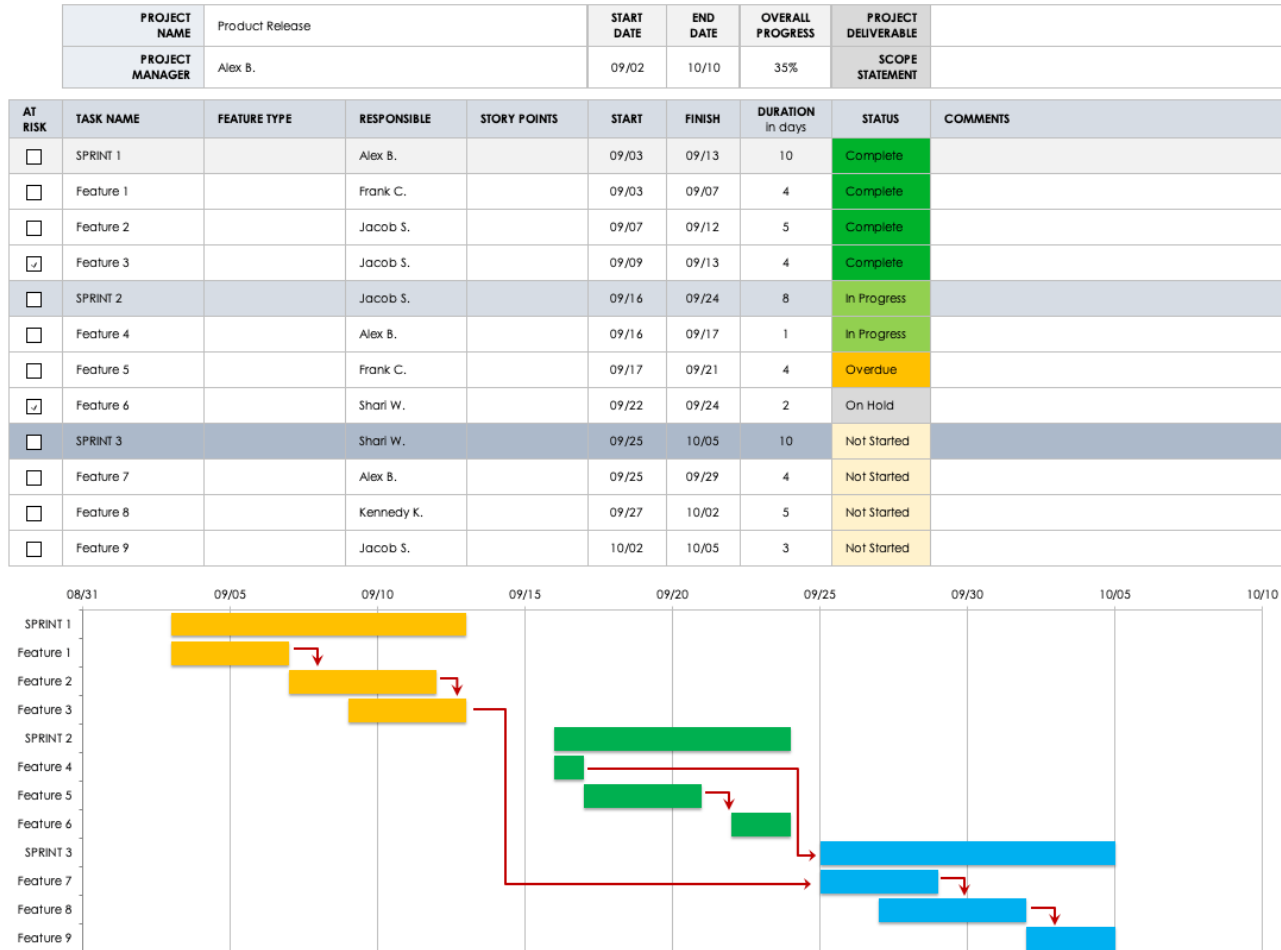
- Graficzne narzędzie do planowania i harmonogramowania projektu
- Paski poziome pokazują czas trwania zadań względem osi czasu
- Umożliwia wizualizację zależności między zadaniami
- Pozwala monitorować postęp realizacji i identyfikować opóźnienia
- Ułatwia komunikację harmonogramu i priorytetów w zespole



530

Przykład

GANTT CHART WITH CRITICAL PATH TEMPLATE



Rys. 11. Źródło:
<https://www.smartsheet.com/content/gantt-chart-critical-path>

Diagram Gantta - kluczowe elementy

- Oś czasu (pozioma) z datami rozpoczęcia i zakończenia zadań
- Lista zadań (pionowa) - kolejność zadań projektu
- Poziome paski - reprezentacja czasu trwania zadań
- Zależności między zadaniami - połączenia/powiązania
- Oznaczenia postępu (% wykonania), kamienie milowe itp.

Zastosowania:

- Planowanie i harmonogramowanie projektu
- Śledzenie postępów prac
- Alokacja zasobów i wykrywanie konfliktów terminów
- Komunikacja z zespołem i interesariuszami
- Identyfikacja miejsca opóźnień i ich wpływu na projekt

Zarządzanie konfliktem w trakcie projektu

- Monitorowanie pojawiających się ryzyk i problemów
- Wczesne wykrywanie konfliktów w zespole
- Stosowanie planów awaryjnych dla ryzyk
- Rozwiązywanie konfliktów przez mediację i negocjacje
- Utrzymanie zespołu zmotywowanego i efektywnego



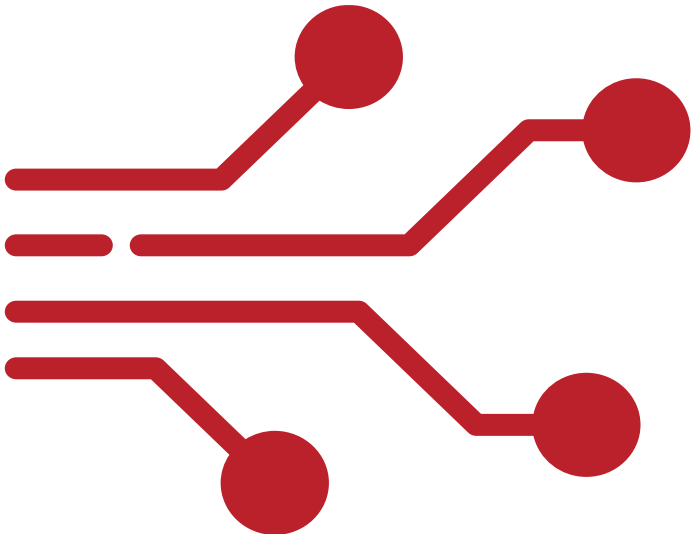
Dobór osobowości do zespołu. Typy osobowości

- Analityk/Myśliciel - logika, planowanie, analiza, przewidywanie ryzyk
- Realizator/Wykonawca - organizacja, dyscyplina, konsekwencja w działaniu
- Innowator/Kreator - generowanie nowych pomysłów, otwartość na zmiany
- Koordynator/Lider - planowanie i kontrola pracy zespołu, podejmowanie decyzji
- Komunikator/Mediator - współpraca, rozwiązywanie konfliktów, motywowanie członków zespołu
- Perfekcjonista/Kontroler jakości - dbałość o szczegóły, weryfikacja zgodności i jakości
- Ekspert techniczny - specjalistyczna wiedza, doradztwo w zakresie merytorycznym
- Animator/Współpracownik - wspieranie integracji zespołu, budowanie relacji
- Optymista/Motywator - utrzymywanie wysokiego morale zespołu, pozytywna energia

534

Piramida potrzeb Maslowa

Poziomy potrzeb

- 
1. Potrzeby fizjologiczne - jedzenie, sen, odpoczynek, energia do pracy
 2. Potrzeby bezpieczeństwa - stabilność zatrudnienia, ochrona, przewidywalność
 3. Potrzeby społeczne/przynależności - współpraca, relacje z zespołem, akceptacja w grupie
 4. Potrzeby uznania/szacunku - poczucie własnej wartości, docenienie kompetencji, status w zespole
 5. Potrzeby samorealizacji - rozwój osobisty, realizacja potencjału, twórczość i innowacyjność

535

Wnioski dla motywowania i krytykowania

Jak motywować członków zespołu?

- Zapewnij podstawowe potrzeby (warunki pracy, bezpieczeństwo, stabilność)
- Wspieraj przynależność i relacje w zespole
- Doceniaj kompetencje i osiągnięcia (uznanie i szacunek)
- Stwarzaj możliwości rozwoju i samorealizacji
- Umożliwiaj angażowanie w kreatywne i odpowiedzialne zadania

Jak udzielać krytyki?

- Skoncentruj się na zachowaniu lub wyniku, nie na osobie
- Zapewnij jasne informacje o oczekiwaniach i standardach
- Połącz krytykę z możliwością poprawy i wsparciem
- Upewnij się, że krytyka nie zagraża poczuciu bezpieczeństwa ani przynależności
- Motywuj do rozwoju, a nie demotywuj

536

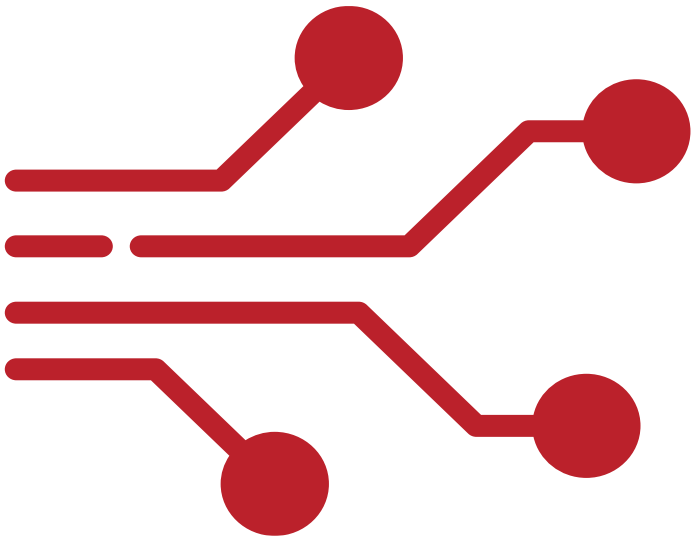
Zarządzanie dokumentacją projektu

- Tworzenie i aktualizacja dokumentów projektowych
- Dokumentowanie decyzji, zmian i ryzyk
- Przechowywanie dokumentacji w centralnym repozytorium
- Udostępnianie dokumentów zespołowi i interesariuszom
- Podstawa do przekazania wiedzy po zakończeniu projektu

537

Zarządzanie jakością projektu

- Definiowanie wymagań jakościowych i standardów
- Monitorowanie zgodności prac z wymaganiami
- Weryfikacja jakości w trakcie realizacji projektu
- Testowanie systemu i przeglądy kodu
- Wprowadzanie działań korygujących w razie odchyleń



538

Normy i standardy jakości w IT

- ISO 9001 - system zarządzania jakością w organizacji
- ISO/IEC 9126/25010 - model jakości oprogramowania (funkcjonalność, niezawodność, użyteczność, wydajność, utrzymywalność, przenośność)
- CMMI (Capability Maturity Model Integration) - model dojrzałości procesów w tworzeniu oprogramowania
- ITIL (Information Technology Infrastructure Library) - dobre praktyki zarządzania usługami IT
- ISO/IEC 27001 - system zarządzania bezpieczeństwem informacji (często powiązany z jakością i niezawodnością)
- Six Sigma/Lean IT - podejście do minimalizacji błędów i optymalizacji procesów

539

Zarządzanie interesariuszami i komunikacja z klientem

- Identyfikacja interesariuszy i ich oczekiwań
- Regularne raportowanie postępów projektu
- Ustalanie priorytetów wymagań i zmian
- Otwartość na feedback i sugestie klienta
- Budowanie zaufania i transparentności w projekcie



Szacowanie ryzyka i plan awaryjny

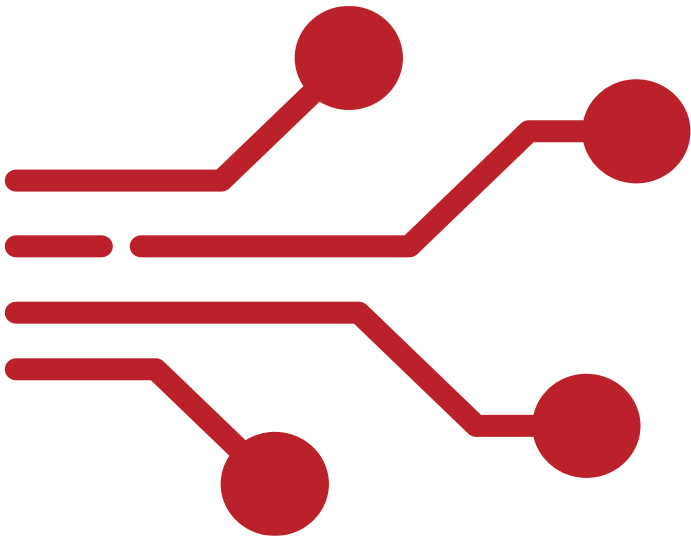
- Identyfikacja potencjalnych zagrożeń dla projektu
- Ocena prawdopodobieństwa i wpływu
- Priorytetyzacja według istotności
- Opracowanie planów reakcji i działań zapobiegawczych
- Ciągłe monitorowanie w trakcie projektu



541

Identyfikacja ryzyk projektu

- Zidentyfikuj wszystkie potencjalne zagrożenia dla projektu
- Przykłady: opóźnienia, błędy techniczne, zmiany wymagań, braki w zasobach
- Określ, które elementy projektu są krytyczne dla sukcesu
- Przypisz odpowiedzialność za monitorowanie ryzyka



542

Ocena ryzyk - prawdopodobieństwo i wpływ

- Oceń prawdopodobieństwo wystąpienia ryzyka: niskie, średnie, wysokie
- Oceń wpływ ryzyka na projekt: niski, średni, wysoki
- Połącz oceny w matrycę ryzyka, aby ustalić priorytety

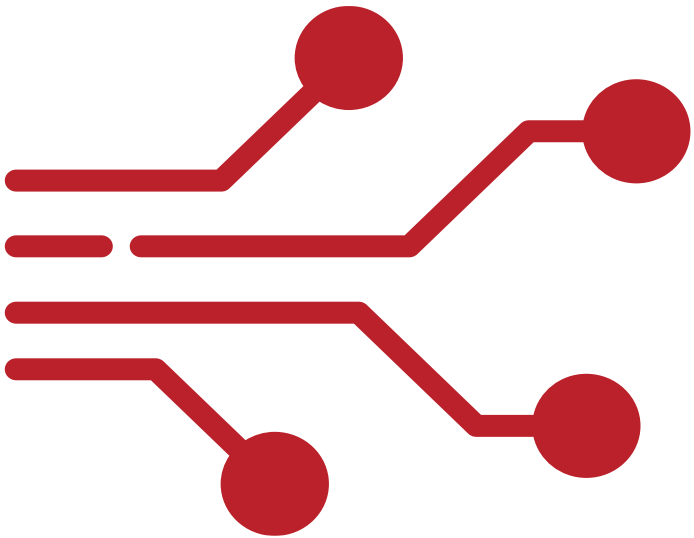
Przykład dla modułu logowania:

Ryzyko: Opóźnienie modułu logowania

Prawdopodobieństwo: Średnie

Wpływ: Wysoki

Poziom ryzyka: Wysokie



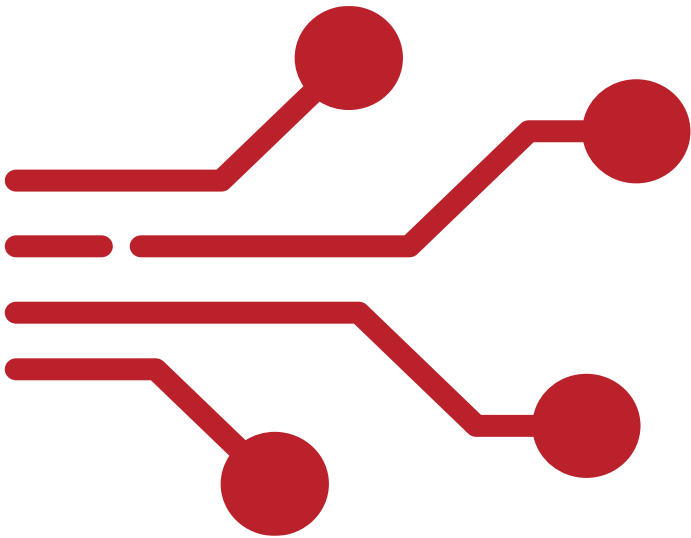
Planowanie reakcji na ryzyko

- Działania zapobiegawcze - minimalizacja prawdopodobieństwa
- Działania łagodzące - zmniejszenie skutków w razie wystąpienia

Przykład: opóźnienie modułu logowania

Zapobiegawcze: przydzielenie doświadczonego programisty

Łagodzące: praca równoległa nad innymi modułami, przesunięcie harmonogramu



Monitorowanie i aktualizacja ryzyk

- Regularnie śledź postęp i potencjalne problemy
- Aktualizuj matrycę ryzyka w trakcie projektu
- Dokumentuj działania podjęte w reakcji na ryzyko
- Przykład: cotygodniowe spotkania statusowe dla modułu logowania



545

Macierz ryzyk

**Prawdopodobieństwo
wysokie**

Wpływ niski

Małe znaczenie

Wpływ średni

Średnie ryzyko

Wpływ wysoki

Krytyczne ryzyko

**Prawdopodobieństwo
średnie**

Małe znaczenie

Średnie ryzyko

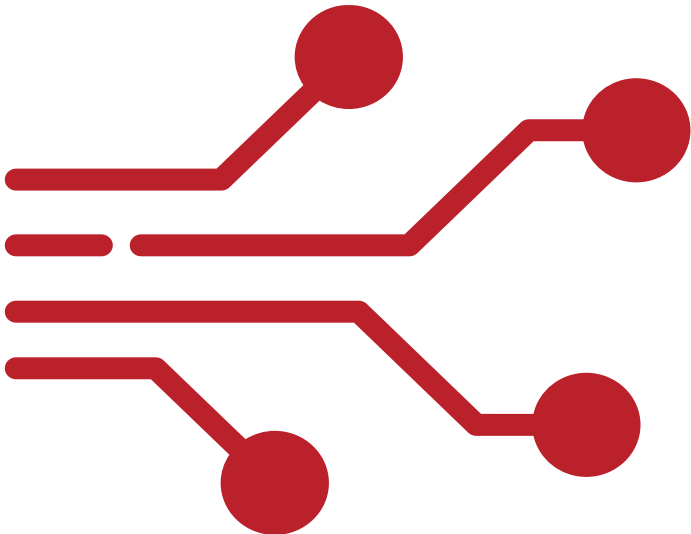
Wysokie ryzyko

**Prawdopodobieństwo
niskie**

Małe znaczenie

Małe znaczenie

Średnie ryzyko



546

Sposoby reakcji na ryzyko

- Unikanie - zmiana planu lub procesu, aby całkowicie wyeliminować ryzyko
- Redukcja/łagodzenie - działania zmniejszające prawdopodobieństwo lub skutki ryzyka
- Przeniesienie - przekazanie ryzyka stronie trzeciej, np. ubezpieczenie, outsourcing
- Akceptacja/poniesienie - świadome przyjęcie ryzyka, gdy koszty działań zapobiegawczych są wyższe niż potencjalna strata
- Eksploatacja - w przypadku ryzyk pozytywnych (szans) - działania zwiększające korzyści

547

Zarządzanie zasobami ludzkimi i motywowanie zespołu

- Przypisanie odpowiednich osób do zadań zgodnie z kompetencjami
- Jasne określenie odpowiedzialności i oczekiwań
- Regularny feedback i wsparcie w rozwiązywaniu problemów
- Motywowanie poprzez uznanie, nagrody i rozwój zawodowy
- Budowanie zaangażowania i współpracy w zespole

548

Zarządzanie konfliktem w zespole projektowym

- Konflikty mogą wynikać z różnic w celach, stylach pracy lub priorytetach
- Wczesne wykrycie konfliktu ułatwia jego rozwiązanie
- Stosowanie mediacji, negocjacji i komunikacji otwartej
- Dokumentowanie ustaleń i decyzji w celu uniknięcia powtórzeń
- Konflikty dobrze zarządzane mogą poprawić współpracę i innowacyjność

549

Zarządzanie zmianą wymagań i zakresu projektu

- Zmiany mogą dotyczyć funkcjonalności, harmonogramu lub kosztów
- Wczesna identyfikacja zmian pozwala ograniczyć o
- Formalny proces zgłaszania i zatwierdzania zmian
- Ocena wpływu zmiany na projekt i zasoby
- Dokumentowanie zmian i komunikacja z zespołem i interesariuszami

550

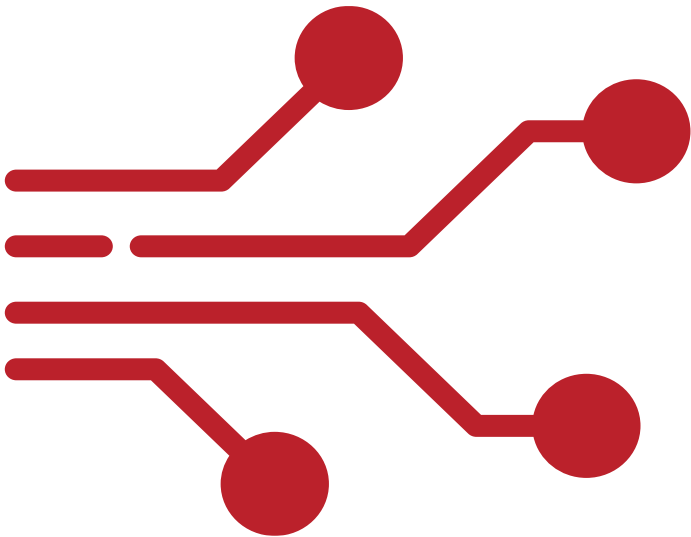
Monitorowanie postępów zadań i kontrola harmonogramu

- Śledzenie realizacji zadań względem harmonogramu
- Identyfikacja opóźnień i przyczyn problemów
- Aktualizacja harmonogramu w razie konieczności
- Regularne raportowanie statusu do kierownika i interesariuszy
- Wykorzystanie narzędzi: wykresy Gantta, tablice Kanban, dashboardy

551

Tablice Kanban

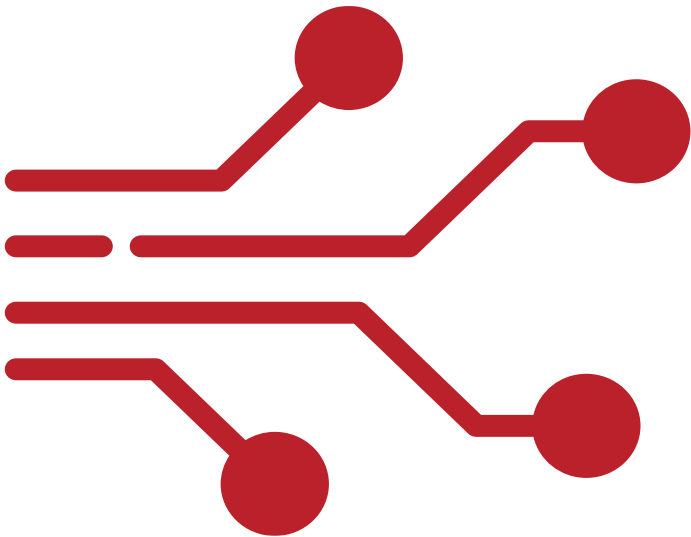
- Wizualne narzędzie do śledzenia postępu prac
- Zadania przedstawione jako karty w kolumnach (np. Do zrobienia, W trakcie, Gotowe)
- Ułatwia identyfikację blokad i wąskich gardeł
- Wspiera samoorganizację zespołu
- Może być używana zarówno w Scrum, jak i innych metodykach Agile



552

Zarządzanie komunikacją w projekcie

- Określenie kanałów komunikacji: e-mail, spotkania, narzędzia online
- Ustalanie częstotliwości i formy raportów
- Jasne przekazywanie celów, zadań i oczekiwań
- Otwartość na feedback i pytania zespołu
- Dokumentowanie kluczowych decyzji i ustaleń



Kontrola kosztów projektu

- Monitorowanie wydatków w stosunku do budżetu
- Identyfikacja nadwyżek i potencjalnych oszczędności
- Analiza kosztów zasobów ludzkich, sprzętu i oprogramowania
- Wprowadzanie działań korygujących w razie przekroczeń
- Regularne raportowanie stanu finansowego projektu



554

Zarządzanie ryzykiem finansowym w projekcie

- Identyfikacja ryzyk finansowych, np. opóźnienia, zmiany kosztów, nieprzewidziane wydatki
- Ocena prawdopodobieństwa i wpływu ryzyk na budżet
- Przygotowanie planów awaryjnych i rezerw finansowych
- Monitorowanie wydatków i porównywanie z planem
- Wdrażanie działań korygujących w przypadku przekroczeń

555

Szacowanie i kontrola jakości w projekcie

- Określenie wymagań jakościowych systemu
- Definiowanie kryteriów akceptacji zadań i etapów
- Monitorowanie zgodności działań z przyjętymi standardami
- Testowanie systemu: jednostkowe, integracyjne, systemowe
- Wdrażanie działań korygujących przy odchyleniach



Zarządzanie dokumentacją jakościową i raportowanie

- Dokumentowanie wyników testów i przeglądów kodu
- Raportowanie błędów i odchyień od standardów
- Tworzenie planów działań korygujących
- Udostępnianie raportów zespołowi i interesariuszom
- Wykorzystanie dokumentacji do nauki na przyszłych projektach



557

Zarządzanie wiedzą i przekazywanie doświadczeń

- Dokumentowanie lekcji wyniesionych z projektu (lessons learned)
- Przechowywanie wiedzy w repozytorium lub wiki projektowym
- Udostępnienie doświadczeń nowym członkom zespołu
- Analiza problemów i sukcesów dla przyszłych projektów
- Tworzenie bazy wiedzy organizacyjnej

558

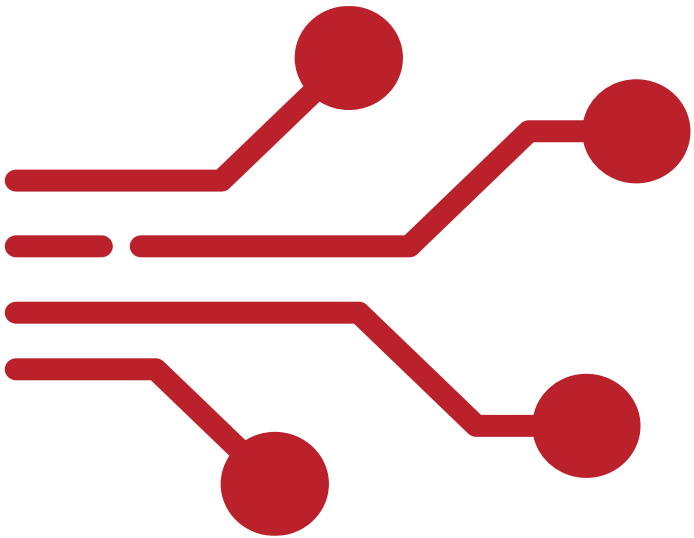
Zamknięcie projektu i ewaluacja końcowa

- Przekazanie systemu lub produktu użytkownikowi końcowemu
- Ocena, czy cele projektu zostały osiągnięte
- Podsumowanie budżetu, harmonogramu i jakości realizacji
- Dokumentowanie wniosków i rekomendacji na przyszłość
- Formalne zakończenie projektu i zwolnienie zasobów

559

Metodyka PMD (Project Management Discipline)

Podejście do zarządzania projektami skupiające się na praktycznych procesach i dyscyplinie w realizacji projektu. Obejmuje planowanie, monitorowanie, kontrolę oraz raportowanie postępów w projekcie, z uwzględnieniem ryzyka i jakości.



560

Cechy metodyki PMD

- Procesy i procedury: Jasne określenie kroków w projekcie - od planowania, przez realizację, po zamknięcie
- Elastyczność: Możliwość dostosowania do małych i średnich projektów, zmiennych wymagań i ograniczonych zasobów
- Skupienie na monitorowaniu: Regularne sprawdzanie postępów, harmonogramów i jakości wykonywanych zadań
- Minimalizacja formalności: Dokumentacja dostosowana do potrzeb zespołu, brak nadmiernej biurokracji
- Łatwość wdrożenia: Proste reguły pozwalają szybko przydzielić role i zadania w zespole

561

Zastosowania:

- Projekty IT o średniej wielkości
- Projekty o dynamicznie zmieniających się wymaganiach
- Zespoły, które potrzebują praktycznego narzędzia do koordynacji pracy bez formalnego nadzoru



562

Metodyka PMBOK (Project Management Body of Knowledge)

PMBOK to międzynarodowy standard zarządzania projektami opracowany przez Project Management Institute (PMI). Zawiera zbiór najlepszych praktyk, procesów i zasad, które pomagają w planowaniu, realizacji i kontroli projektów w sposób ustrukturyzowany i powtarzalny

563

Pięć faz projektu w PMBOK

1. Inicjacja: Określenie celów, interesariuszy, wstępnego zakresu i budżetu
2. Planowanie: Szczegółowy plan projektu, harmonogram, zarządzanie ryzykiem, alokacja zasobów
3. Wykonanie: Realizacja zadań, koordynacja zespołu, komunikacja z interesariuszami
4. Monitorowanie i kontrola: Śledzenie postępów, kontrola jakości, kosztów, harmonogramu i ryzyk
5. Zamknięcie: Przekazanie produktu, analiza doświadczeń, dokumentacja i ewaluacja projektu

564

Obszary wiedzy

1. Integracja projektu
2. Zakres projektu
3. Czas projektu
4. Koszty projektu
5. Jakość projektu
6. Zasoby ludzkie
7. Komunikacja
8. Ryzyko
9. Zamówienia/dostawcy
10. Interesariusze



Zalety PMBOK

- Globalny standard, uznany w certyfikacji PMP
- Kompletna struktura procesów i ról w projekcie
- Ułatwia standaryzację projektów w dużych organizacjach

Wady PMBOK

- Może być zbyt formalny i biurokratyczny dla małych projektów
- Wymaga dużej ilości dokumentacji i raportowania

566

Dziękuję za uwagę

Wykład nr 12

Temat: Metodyki zwinne. Diagram spalania

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

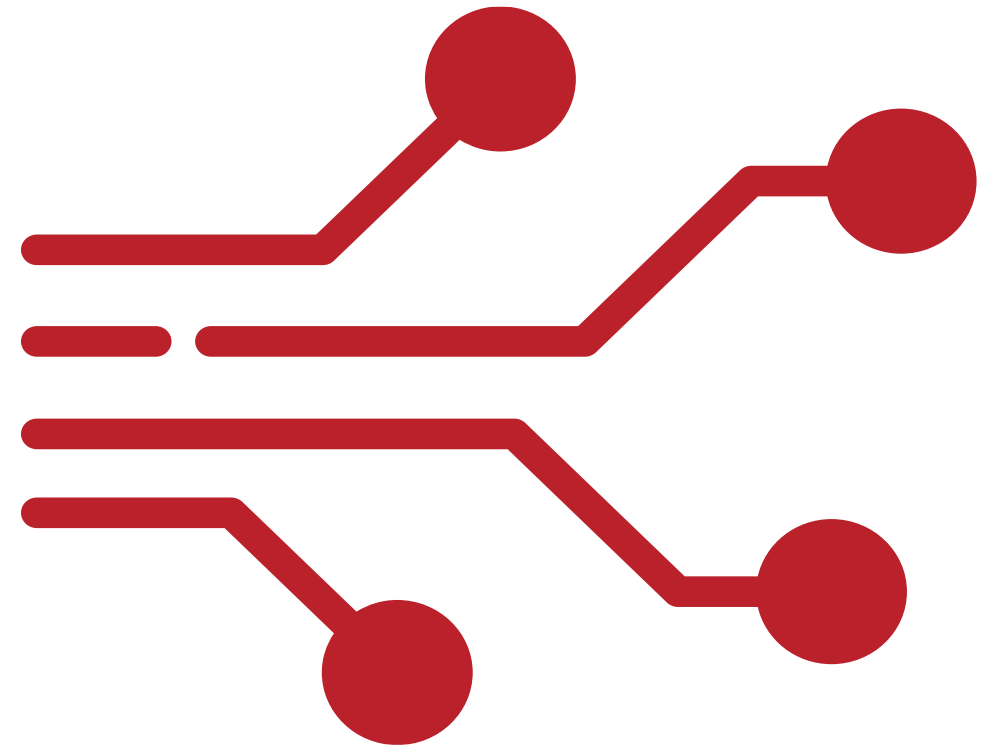
Plan prezentacji

- Wprowadzenie
- Metodyki zwinne



569

Wprowadzenie



570

Dlaczego powstały metodyki zwinne

- Tradycyjne podejścia były mało elastyczne
- Zmiany wymagań pojawiały się w trakcie projektu
- Klienci oczekiwali szybszego dostarczania efektów
- Potrzebne było częstsze testowanie i doskonalenie produktu
- Ważna stała się bliska współpraca zespołu z klientem



571

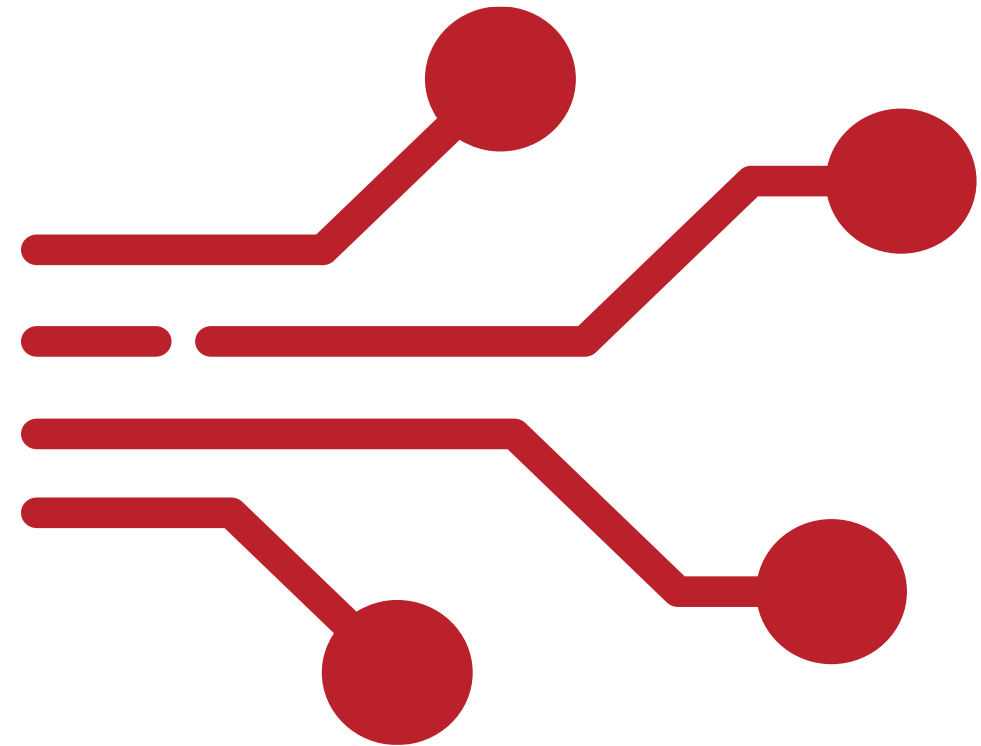
Manifest Agile

- Ludzie i interakcje ponad procesy i narzędzia
- Działające oprogramowanie ponad obszerną dokumentację
- Współpraca z klientem ponad negocjacje kontraktów
- Reagowanie na zmiany ponad realizację planu



572

Metodyki zwinne



573

Cechy podejścia zwinnego

- Rozwój oprogramowania w krótkich iteracjach
- Częste dostarczanie działających fragmentów systemu
- Stała współpraca zespołu z klientem
- Gotowość do wprowadzania zmian w projekcie
- Silny nacisk na komunikację w zespole



574

Agile a podejście tradycyjne (Waterfall)

- Waterfall zakłada realizację projektu w kolejnych etapach
- Agile opiera się na iteracjach i stopniowym rozwoju systemu
- W Waterfall zmiany wymagań są trudne do wprowadzenia
- Agile zakłada możliwość ciągłego doprecyzowywania wymagań
- Agile umożliwia szybsze dostarczanie działających funkcji

575

Iteracyjny rozwój oprogramowania

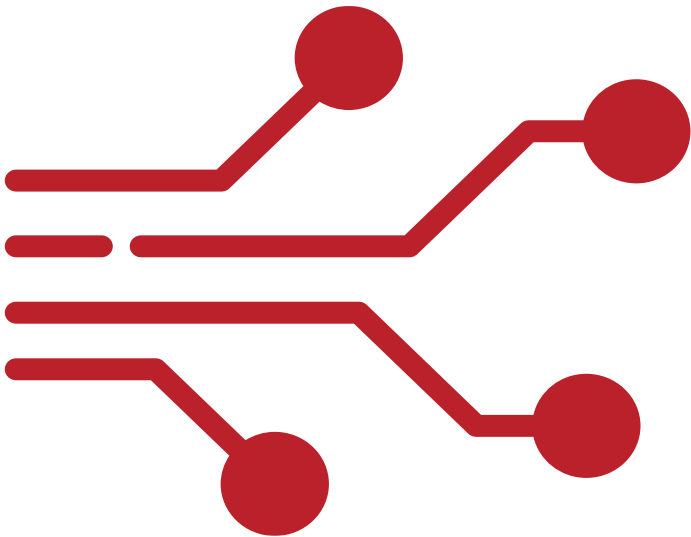
- System rozwijany jest w kolejnych iteracjach
- Każda iteracja obejmuje analizę, implementację i testowanie
- Iteracje są stosunkowo krótkie (np. 1-4 tygodnie)
- Po każdej iteracji powstaje działający fragment systemu
- Pozwala to szybciej uzyskać informacje zwrotne od użytkowników



576

Inkrementalne dostarczanie funkcjonalności

- System rozwijany jest poprzez kolejne przyrosty funkcjonalności
- Każdy przyrost rozszerza możliwości systemu
- Użytkownicy mogą wcześniej korzystać z części systemu
- Zespół może szybciej wykrywać problemy projektowe
- Ułatwia stopniowy rozwój produktu



577

Praca w krótkich iteracjach

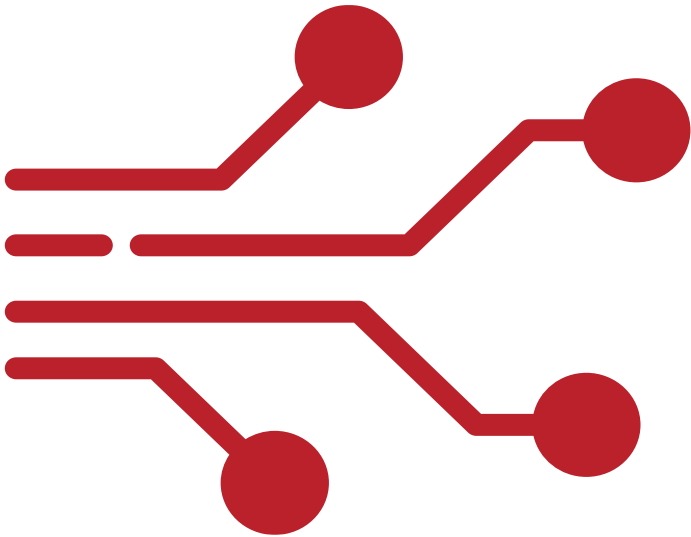
- Prace nad systemem dzielone są na krótkie cykle
- Iteracje trwają zazwyczaj od 1 do 4 tygodni
- Każda iteracja kończy się działającą wersją systemu
- Zespół regularnie ocenia postęp prac
- Umożliwia szybkie reagowanie na zmiany



578

Współpraca z klientem w Agile

- Klient aktywnie uczestniczy w procesie wytwarzania systemu
- Wymagania mogą być doprecyzowywane w trakcie projektu
- Klient regularnie ocenia kolejne wersje systemu
- Informacja zwrotna wpływa na dalszy rozwój produktu
- Zmniejsza ryzyko powstania systemu niespełniającego oczekiwań



Czym jest Scrum?

- Scrum jest frameworkiem zarządzania pracą zespołu
- Jest jednym z najpopularniejszych podejść Agile
- Opiera się na pracy w iteracjach zwanych sprintami
- Zakłada regularne planowanie, przegląd i usprawnianie pracy
- Umożliwia elastyczne rozwijanie produktu

580

Role w Scrum

- Product Owner
- Scrum Master
- Zespół deweloperski
- Każda rola ma jasno określone zadania
- Role wspólnie odpowiadają za rozwój produktu



581

Product Owner

- Reprezentuje interesy klienta i użytkowników
- Określa priorytety rozwoju produktu
- Zarządza listą zadań w Product Backlog
- Decyduje o kolejności realizacji funkcjonalności
- Odpowiada za maksymalizację wartości produktu

582

Scrum Master

- Wspiera zespół w stosowaniu zasad Scrum
- Pomaga usuwać przeszkody utrudniające pracę zespołu
- Dbą o prawidłowy przebieg procesu Scrum
- Wspiera komunikację w zespole projektowym
- Pomaga zespołowi doskonalić sposób pracy



583

Zespół deweloperski

- Odpowiada za implementację funkcjonalności systemu
- Składa się z programistów, testerów i innych specjalistów
- Jest zespołem samoorganizującym się
- Wspólnie odpowiada za realizację zadań w sprincie
- Dostarcza działający fragment produktu na koniec iteracji

584

Product Backlog

- Lista wszystkich funkcjonalności systemu
- Zawiera wymagania dotyczące produktu
- Elementy backlogu są uporządkowane według priorytetu
- Jest stale aktualizowany w trakcie projektu
- Zarządzany przez Product Ownera



585

User Stories

- Krótkie opisy funkcjonalności systemu z perspektywy użytkownika
- Zawierają informacje o celu i oczekiwanym rezultacie
- Ułatwiają komunikację między zespołem a klientem
- Pozwalają określić priorytety w Product Backlog
- Podstawowy element planowania sprintu



Sprint Backlog

- Lista zadań wybranych do realizacji w konkretnym sprincie
- Tworzony przez zespół deweloperski na początku iteracji
- Zawiera szczegółowe kroki potrzebne do implementacji funkcjonalności
- Jest aktualizowany w trakcie sprintu
- Pozwala monitorować postęp prac w iteracji

587

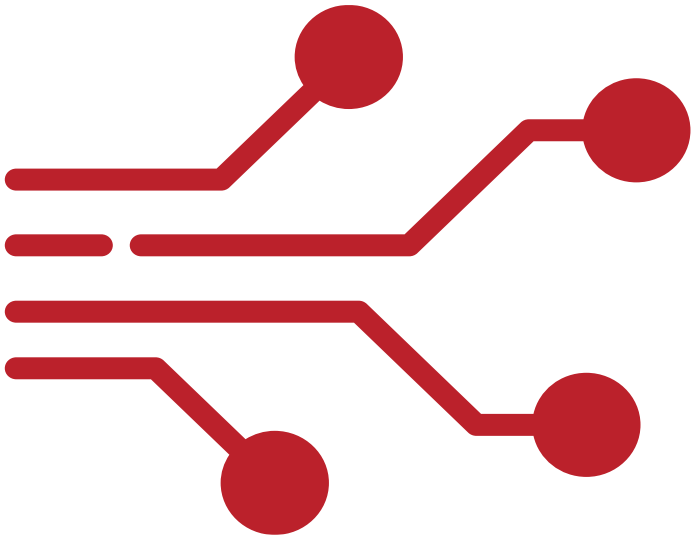
Definition of Done (DoD)

- Zestaw kryteriów określających, kiedy zadanie lub funkcjonalność jest ukończona
- Ustala wspólne standardy jakości w zespole
- Zapewnia, że dostarczony fragment systemu jest działający i przetestowany
- Pomaga uniknąć nieporozumień między członkami zespołu
- Stosowane dla każdej user story w sprincie

588

Sprint

- Krótki, czasowo ograniczony cykl pracy zespołu
- Zazwyczaj trwa od 1 do 4 tygodni
- Kończy się dostarczeniem działającego fragmentu produktu
- Podczas sprintu zespół realizuje wybrane elementy Product Backlog
- Podstawowy element iteracyjnego rozwoju w Scrum



589

Sprint Planning

- Spotkanie rozpoczynające każdy sprint
- Zespół wybiera elementy Product Backlog do realizacji
- Określa się cel sprintu (Sprint Goal)
- Zespół dzieli wybrane elementy na konkretne zadania
- Ustala się priorytety i plan pracy na iterację



590

Daily Scrum

- Krótkie, codzienne spotkanie zespołu (ok. 15 minut)
- Każdy członek zespołu odpowiada na trzy pytania:
 1. Co zrobiłem wczoraj?
 2. Co zrobię dzisiaj?
 3. Jakie przeszkody napotkałem?
- Umożliwia bieżące monitorowanie postępów
- Pomaga zespołowi szybko reagować na problemy
- Wspiera komunikację i samoorganizację zespołu

591

Sprint Review

- Spotkanie podsumowujące zakończony sprint
- Zespół prezentuje działające funkcjonalności klientowi i interesariuszom
- Omawiane są osiągnięcia i ewentualne problemy
- Zbierana jest informacja zwrotna od klienta
- Umożliwia dostosowanie kolejnych iteracji



592

Sprint Retrospective

- Spotkanie zespołu po zakończonym sprincie
- Analiza przebiegu sprintu i procesu pracy
- Identyfikacja problemów i przeszkód
- Wypracowanie usprawnień do kolejnych iteracji
- Wspiera ciągłe doskonalenie zespołu



593

Estymacja zadań

- Określanie trudności i czasu potrzebnego na realizację zadań
- Pozwala planować sprinty i iteracje
- W Scrumie najczęściej stosuje się story points zamiast godzin
- Szacowanie wykonuje cały zespół deweloperski
- Ułatwia przewidywanie tempa pracy zespołu (velocity)

594

Story Points

- Jednostka szacowania zadań w Agile
- Określa względną złożoność i wysiłek potrzebny do realizacji
- Pozwala porównywać zadania między sobą
- Nie mierzy bezpośrednio czasu w godzinach
- Wspiera planowanie sprintów i przewidywanie velocity

595

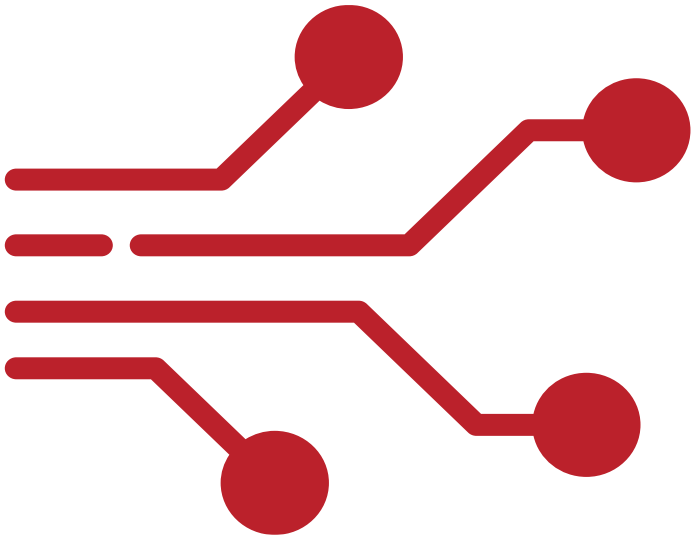
Planning Poker

- Technika szacowania zadań w Scrum
- Każdy członek zespołu ocenia zadanie niezależnie kartami z wartościami punktów
- Po ujawnieniu wszystkich kart następuje dyskusja o różnicach w ocenie
- Celem jest wypracowanie konsensusu co do złożoności zadania
- Wspiera dokładniejsze i bardziej świadome estymacje

596

Diagram spalania (Burndown Chart)

- Graficzne przedstawienie postępu prac w sprincie
- Pokazuje ilość pracy pozostałej do wykonania w czasie
- Oś X: czas trwania sprintu
- Oś Y: pozostałe zadania lub story points
- Umożliwia szybkie zidentyfikowanie opóźnień
- Wspiera monitorowanie tempa pracy zespołu (velocity)



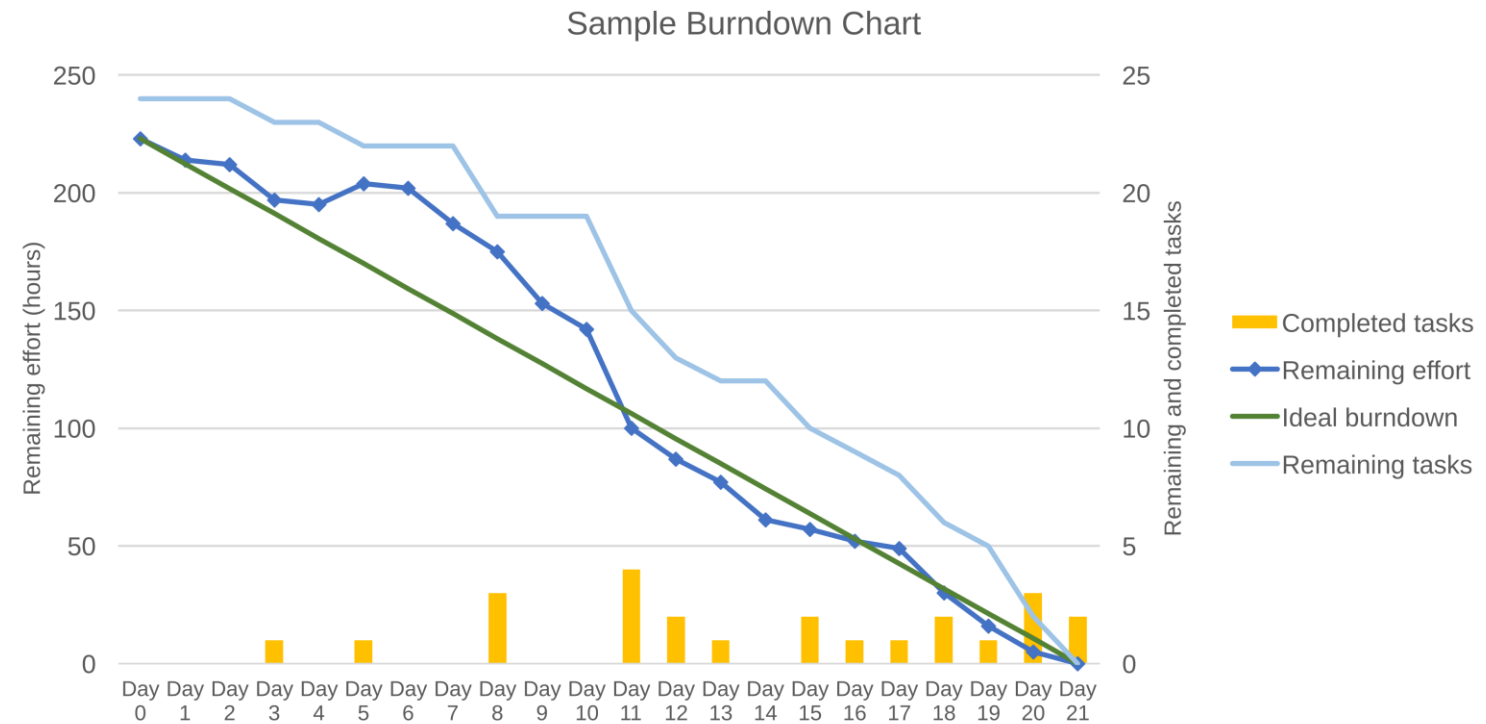
597

Interpretacja diagramu spalania

- Linia idealna - planowany spadek pracy w sprincie
- Linia rzeczywista - faktyczny postęp zespołu
- Nad linią idealną - prace opóźnione
- Pod linią idealną - zespół wyprzedza plan
- Analiza pozwala korygować tempo pracy i planowanie kolejnych sprintów



Przykład



Rys. 12. Źródło: Wikipedia

599

Narzędzia wspierające Agile

- Jira
- Azure DevOps
- GitLab
- Trello
- Pozwalają zarządzać backlogiem, sprintami i zadaniami
- Umożliwiają wizualizację postępu i raportowanie

600

Zalety metodyk zwinnych

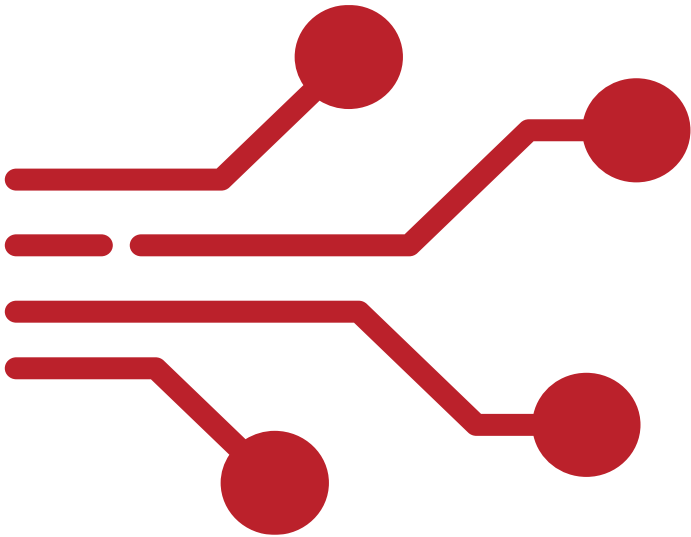
- Szybsze dostarczanie działających funkcjonalności
- Lepsze dopasowanie systemu do potrzeb użytkownika
- Regularna informacja zwrotna od klienta
- Większa elastyczność w reagowaniu na zmiany
- Zwiększona motywacja i współpraca zespołu



601

Ograniczenia Agile

- Wymaga pełnego zaangażowania klienta
- Może być trudne w dużych, rozproszonych zespołach
- Nie zawsze łatwo przewidzieć dokładny harmonogram i koszty
- Wymaga dojrzałego i doświadczonego zespołu
- Nie eliminuje ryzyka błędów, wymaga ciągłej kontroli jakości



602

Agile i cyberbezpieczeństwo

- Umożliwia szybkie reagowanie na nowe zagrożenia
- Pozwala wprowadzać poprawki i aktualizacje w krótkich iteracjach
- Regularne testy i przeglądy zwiększają bezpieczeństwo systemu
- Wspiera współpracę zespołów deweloperskich i bezpieczeństwa
- Ułatwia integrację działań związanych z zarządzaniem ryzykiem

603

Dziękuję za uwagę



Wykład nr 13

Temat: Systemy kontroli wersji

Paweł Karczmarek
Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

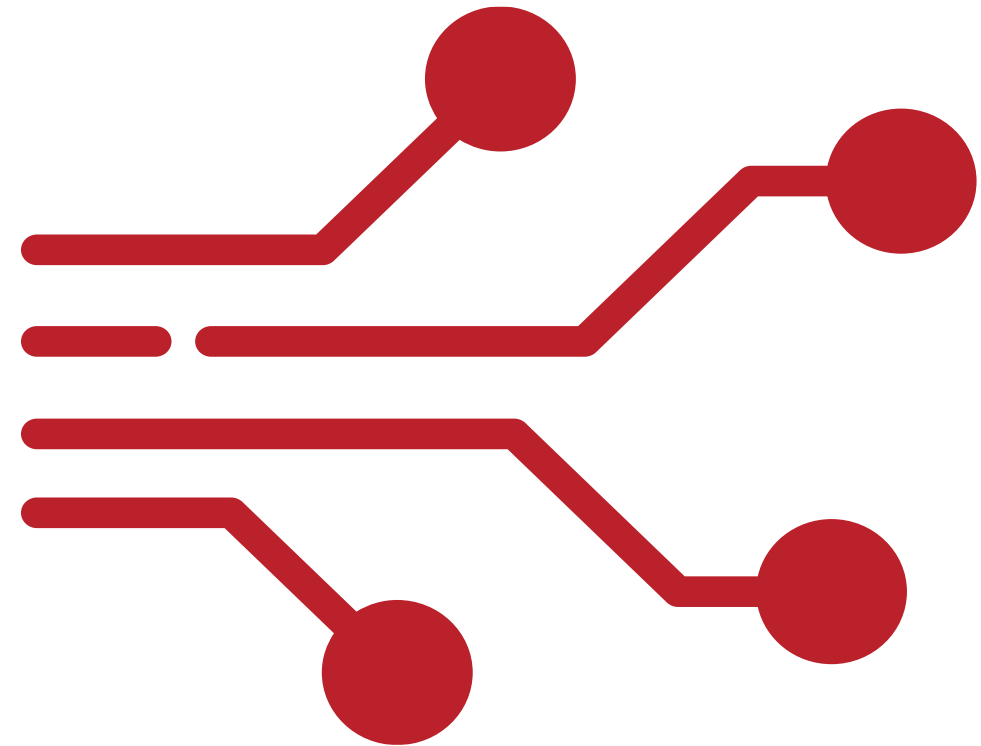
Plan prezentacji

- Wprowadzenie
- Git



606

Wprowadzenie



607

Systemy kontroli wersji

- Umożliwiają śledzenie zmian w kodzie źródłowym
- Pozwalają przechowywać historię rozwoju projektu
- Ułatwiają współpracę wielu programistów
- Pozwalają przywracać wcześniejsze wersje plików
- Wspierają zarządzanie rozwojem oprogramowania



Dlaczego kontrola wersji jest potrzebna?

- Pozwala śledzić zmiany w projekcie
- Umożliwia pracę wielu programistów nad jednym systemem
- Zapobiega utracie kodu źródłowego
- Ułatwia cofanie błędnych zmian
- Wspiera organizację pracy zespołu



Problemy pracy bez systemu kontroli wersji

- Trudność w śledzeniu zmian w kodzie
- Ryzyko nadpisywania pracy innych programistów
- Brak historii rozwoju projektu
- Trudności w przywracaniu wcześniejszych wersji plików
- Większe ryzyko utraty kodu źródłowego

610

Podstawowe pojęcia systemów kontroli wersji

- Repozytorium
- Commit
- Branch (gałąź)
- Merge (scalanie zmian)
- Historia zmian projektu



611

Lokalne systemy kontroli wersji

- Przechowują historię zmian na jednym komputerze
- Zarządzają kolejnymi wersjami plików
- Nie wymagają połączenia z serwerem
- Są przeznaczone głównie dla pojedynczego programisty
- Mają ograniczone możliwości pracy zespołowej



612

Centralne systemy kontroli wersji

- Wykorzystują centralny serwer repozytorium
- Programiści pobierają pliki z serwera
- Zmiany są zapisywane w centralnym repozytorium
- Ułatwiają współpracę zespołową
- Wymagają dostępu do serwera



613

Rozproszone systemy kontroli wersji

- Każdy programista posiada pełną kopię repozytorium
- Historia projektu jest dostępna lokalnie
- Możliwa jest praca bez połączenia z siecią
- Ułatwiają równoległą pracę wielu programistów
- Są obecnie najczęściej stosowanym rozwiązaniem



614

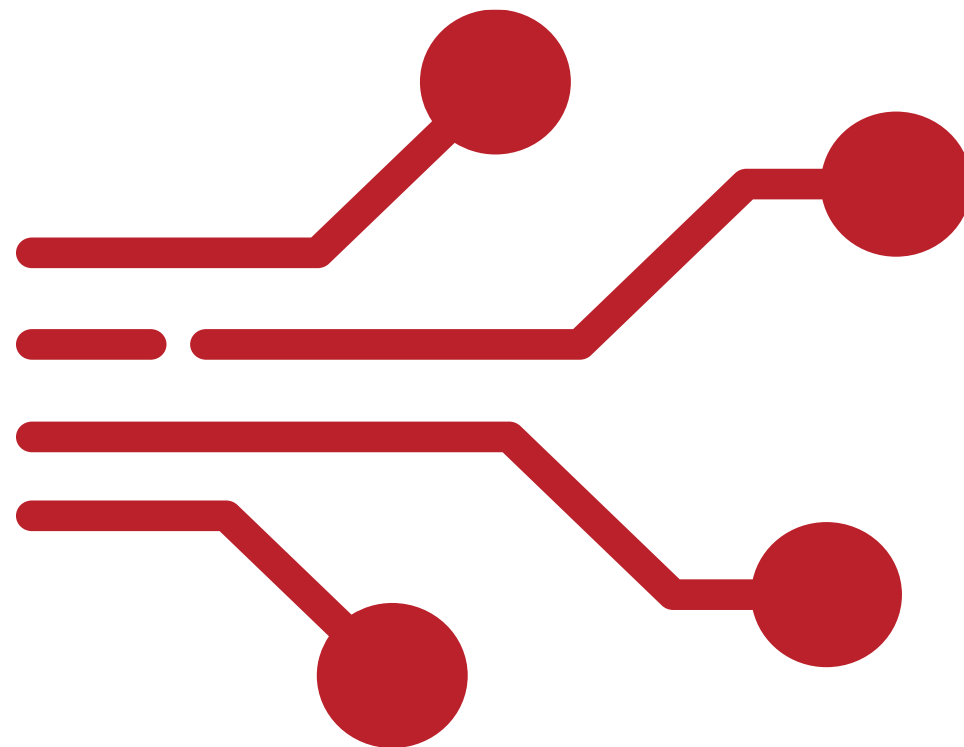
Porównanie systemów centralnych i rozproszonych

- Różnią się sposobem przechowywania repozytorium
- Systemy centralne posiadają jedno główne repozytorium
- Systemy rozproszone przechowują pełną kopię repozytorium u każdego użytkownika
- Systemy rozproszone umożliwiają pracę offline
- Systemy rozproszone są obecnie standardem w projektach informatycznych



615

Git



616

Wprowadzenie do Git

- Git jest rozproszonym systemem kontroli wersji
- Został zaprojektowany do zarządzania kodem źródłowym
- Umożliwia śledzenie historii zmian projektu
- Wspiera równoległą pracę wielu programistów
- Jest obecnie standardem w branży IT



617

Model pracy w Git

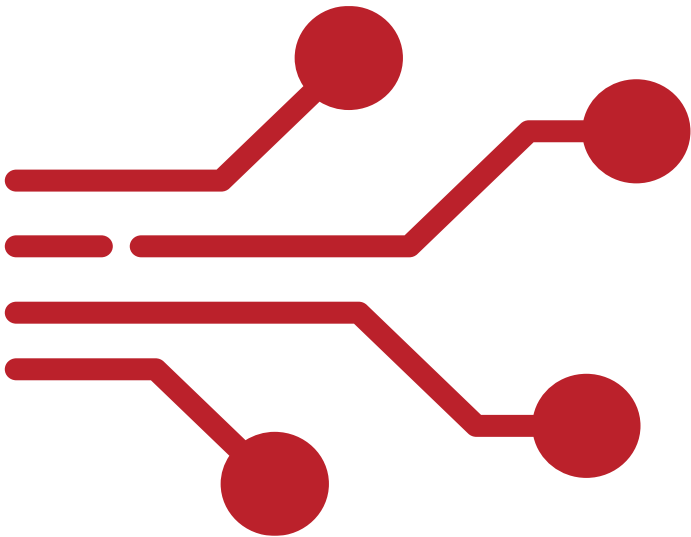
- Programista pracuje na lokalnej kopii repozytorium
- Zmiany są zapisywane lokalnie jako commity
- Repozytorium lokalne może być synchronizowane z repozytorium zdalnym
- Zmiany są przesyłane na serwer lub pobierane z serwera
- Wspiera równoległą pracę wielu programistów



618

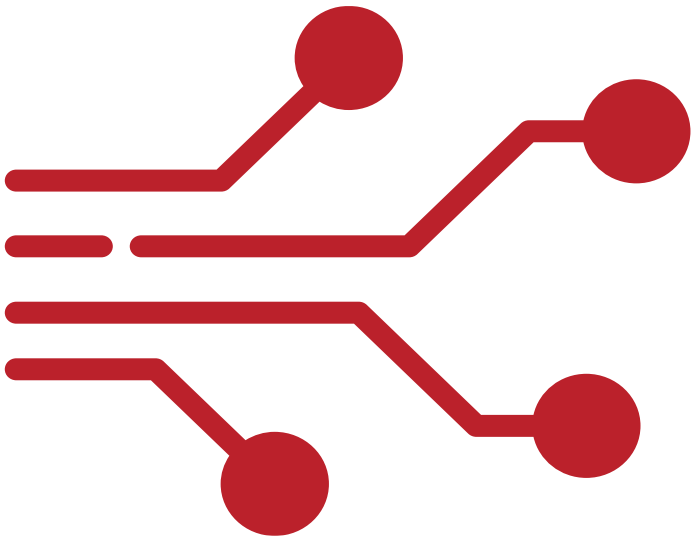
Obszar roboczy, staging area i repozytorium

- Obszar roboczy (working directory)
- Staging area (obszar przygotowania zmian)
- Repozytorium lokalne
- Zmiany przechodzą przez kolejne etapy zapisu
- Pozwala kontrolować, które zmiany trafią do commitów



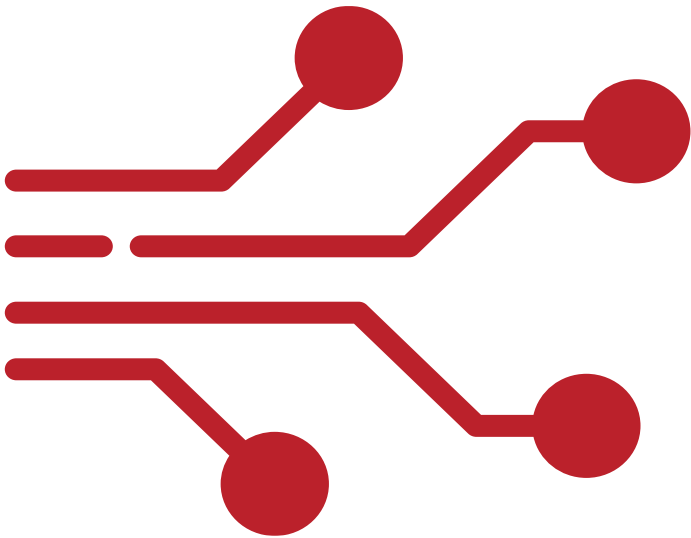
Najważniejsze operacje w Git

- git init - utworzenie repozytorium
- git add - dodanie zmian do staging area
- git commit - zapisanie zmian w repozytorium
- git status - sprawdzenie stanu repozytorium
- git log - przeglądanie historii zmian



Inicjalizacja repozytorium

- Repozytorium można utworzyć w istniejącym katalogu projektu
- Git zaczyna śledzić zmiany w plikach
- Tworzony jest ukryty katalog .git
- Repozytorium zawiera historię zmian projektu
- Jest to pierwszy krok pracy z Git



621

Dodawanie plików i tworzenie commitów

- Zmodyfikowane pliki trafiają do obszaru roboczego
- Polecenie git add dodaje zmiany do staging area
- Polecenie git commit zapisuje zmiany w repozytorium
- Commit zawiera opis wprowadzonych zmian
- Każdy commit tworzy nową wersję projektu



622

Historia zmian w projekcie

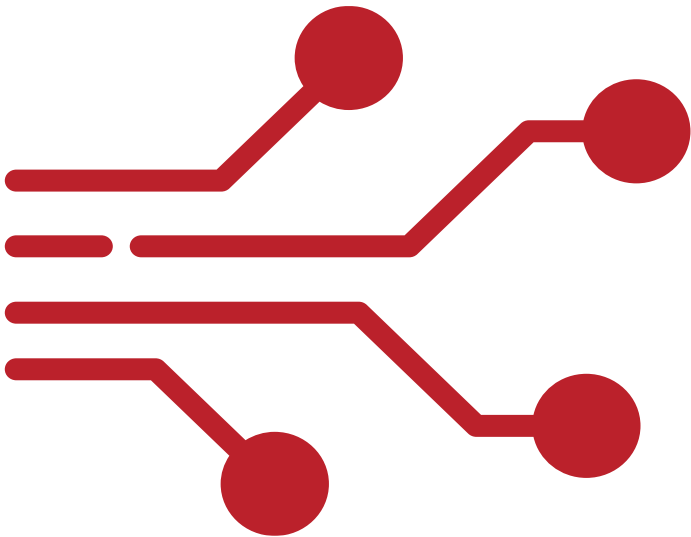
- Polecenie git log pokazuje listę commitów
- Każdy commit zawiera autora, datę i opis zmian
- Pozwala śledzić rozwój projektu krok po kroku
- Umożliwia analizę wprowadzonych modyfikacji
- Pomaga identyfikować przyczyny błędów



623

Przywracanie wcześniejszych wersji plików

- Git pozwala wrócić do wcześniejszej wersji pliku lub całego projektu
- Polecenia: git checkout, git restore
- Umożliwia cofnięcie błędnych zmian
- Pomaga odzyskać poprzednie stabilne wersje
- Wspiera bezpieczeństwo pracy nad kodem



624

Repozytoria zdalne

- Repozytorium może znajdować się na serwerze
- Umożliwia współpracę wielu programistów
- Przechowuje centralną kopię projektu
- Ułatwia synchronizację zmian między użytkownikami
- Jest często hostowane na platformach takich jak GitLab lub GitHub



625

Klonowanie repozytorium

- Umożliwia pobranie projektu z repozytorium zdalnego
- Tworzy lokalną kopię repozytorium
- Zawiera pełną historię zmian projektu
- Wykonywane poleceniem git clone
- Jest często pierwszym krokiem pracy z istniejącym projektem



626

Wysyłanie i pobieranie zmian

- git push - wysyłanie zmian do repozytorium zdalnego
- git pull - pobieranie zmian z repozytorium zdalnego
- git fetch - pobieranie zmian bez automatycznego scalania
- Synchronizacja pracy wielu programistów
- Aktualizacja lokalnego repozytorium



627

Współpraca wielu programistów nad jednym projektem

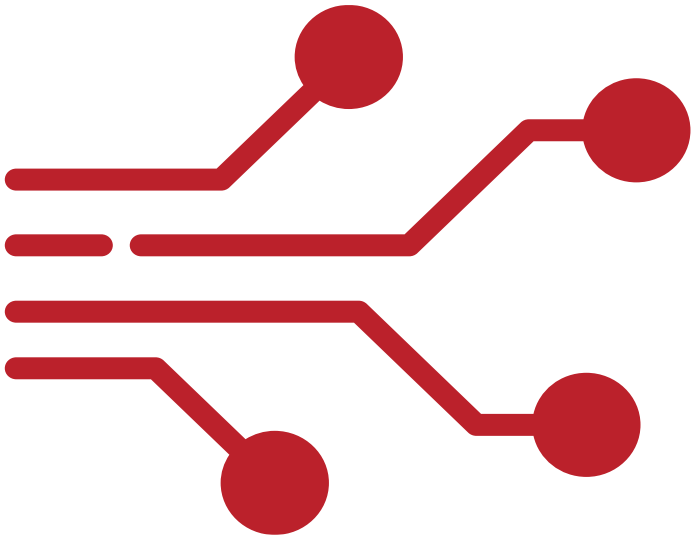
- Każdy programista pracuje na własnej kopii repozytorium
- Zmiany są zapisywane jako commity
- Programiści synchronizują zmiany z repozytorium zdalnym
- Wykorzystywane są gałęzie do równoległego rozwoju funkcjonalności
- Zmiany są integrowane z główną wersją projektu



628

Gałęzie w systemie kontroli wersji

- Gałąź (branch) to niezależna linia rozwoju projektu
- Pozwala rozwijać nowe funkcjonalności bez wpływu na główną wersję kodu
- Umożliwia równoległą pracę nad różnymi elementami systemu
- Każda gałąź posiada własną historię commitów
- Najczęściej używaną gałęzią jest gałąź główna (np. main lub master)



629

Tworzenie i przełączanie gałęzi

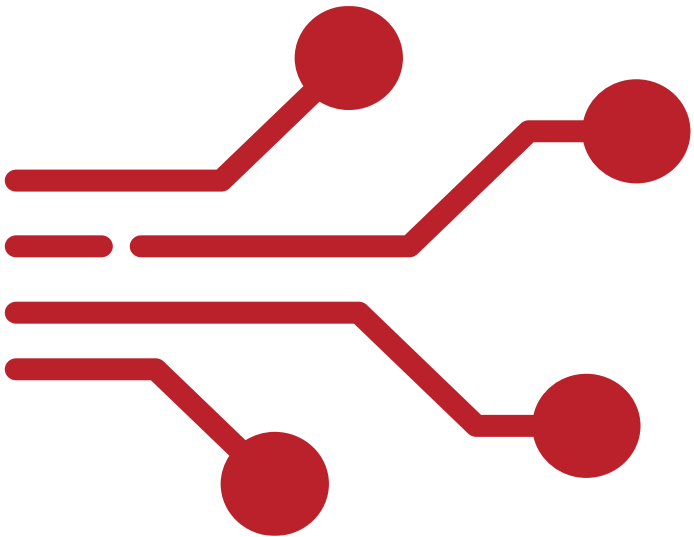
- git branch - wyświetlanie i tworzenie gałęzi
- git checkout - przełączanie między gałęziami
- git switch - nowoczesne polecenie przełączania gałęzi
- Umożliwia pracę nad różnymi funkcjonalnościami projektu
- Pozwala izolować zmiany w kodzie



630

Scalanie zmian

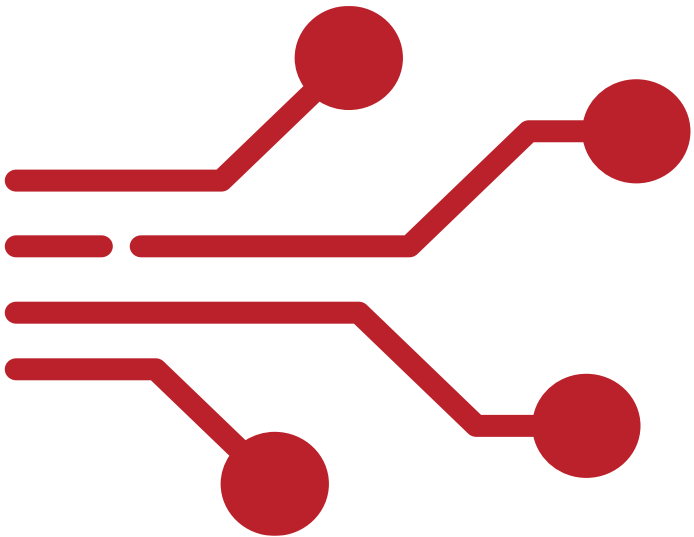
- Scalanie łączy zmiany z różnych gałęzi projektu
- Wykonywane poleceniem git merge
- Integruje nową funkcjonalność z główną gałęzią
- Zachowuje historię commitów
- Jest ważnym elementem pracy zespołowej



631

Konflikty przy scalaniu i ich rozwiązywanie

- Konflikt powstaje, gdy różne zmiany dotyczą tego samego fragmentu kodu
- Git nie może automatycznie połączyć zmian
- Programista musi ręcznie zdecydować, którą wersję zachować
- Po rozwiązaniu konfliktu tworzony jest commit scalający
- Konflikty są naturalnym elementem pracy zespołowej



632

Jak tworzyć dobre commity?

- Commit powinien dotyczyć jednej zmiany lub funkcjonalności
- Opis commitu powinien być krótki i zrozumiały
- Commity powinny być wykonywane regularnie
- Należy unikać bardzo dużych commitów
- Historia commitów powinna być czytelna



633

Strategie pracy na gałęziach

- Gałąź główna zawiera stabilną wersję systemu
- Nowe funkcjonalności rozwijane są w osobnych gałęziach
- Po zakończeniu prac zmiany są scalane z gałęzią główną
- Poprawki błędów mogą być rozwijane w osobnych gałęziach
- Ułatwia organizację pracy zespołu



634

Code review, pull request i merge request

- Code review to przegląd zmian w kodzie przez innych programistów
- Pull request/merge request służy do zgłaszania zmian w repozytorium
- Zmiany są analizowane przed ich włączeniem do głównej gałęzi
- Umożliwia komentowanie fragmentów kodu
- Wspiera poprawę jakości oprogramowania

635

Proces tworzenia funkcjonalności w GitLab

1. Utworzenie zgłoszenia (issue)
2. Utworzenie gałęzi do zadania
3. Implementacja zmian i commity
4. Utworzenie merge request
5. Code review i poprawki
6. Scalanie zmian do gałęzi głównej



Typowy workflow

- Issue (zadanie)
- Utworzenie branch feature/issue-42
- Implementacja kodu + commity
- git push
- Merge Request
- Code Review - poprawki - commit – push
- Akceptacja MR
- Merge do main/master
- Kod trafia do głównej wersji systemu



637

Ważne komendy Git

- git clone - pobranie repozytorium z serwera
- git pull - pobranie najnowszych zmian z repozytorium
- git status - sprawdzenie stanu repozytorium
- git add - dodanie zmian do staging area
- git commit - zapisanie zmian w historii projektu
- git push - wysłanie zmian do repozytorium zdalnego
- git branch - tworzenie i wyświetlanie gałęzi
- git switch/git checkout - przełączanie gałęzi
- git merge - scalanie zmian z różnych gałęzi
- git log - przeglądanie historii commitów

638

Najczęstsze błędy przy pracy z kontrolą wersji

- Brak regularnych commitów
- Nieczytelne opisy commitów
- Wprowadzanie zmian bez użycia gałęzi
- Konflikty wynikające z rzadkiej synchronizacji repozytorium
- Przechowywanie plików tymczasowych w repozytorium



639

Śledzenie zmian i odpowiedzialność za kod

- Każda zmiana w repozytorium jest zapisana w historii
- Commit zawiera autora oraz opis zmiany
- Umożliwia identyfikację osób wprowadzających zmiany
- Pozwala analizować rozwój projektu
- Wspiera odpowiedzialność za jakość kodu



640

Kontrola wersji a bezpieczeństwo projektu

- Pozwala śledzić wszystkie zmiany w kodzie
- Umożliwia analizę wprowadzonych modyfikacji
- Pomaga wykrywać nieautoryzowane zmiany
- Wspiera audyt bezpieczeństwa systemu
- Ułatwia analizę incydentów bezpieczeństwa

641

Kopie bezpieczeństwa i odzyskiwanie projektu

- Repozytorium przechowuje pełną historię projektu
- Repozytoria zdalne stanowią dodatkową kopię kodu
- Umożliwia odzyskanie wcześniejszych wersji plików
- Chroni projekt przed utratą danych
- Wspiera ciągłość pracy zespołu



642

Kontrola wersji w DevOps i CI/CD

- Repozytorium Git jest centralnym elementem procesu DevOps
- Zmiany w kodzie mogą automatycznie uruchamiać proces budowania
- Systemy CI/CD wykonują testy po każdym commicie
- Automatyzują proces wdrażania aplikacji
- Wspierają ciągłą integrację i rozwój systemu



643

GitHub, GitLab i Bitbucket

- Platformy hostingowe dla repozytoriów Git
- Umożliwiają przechowywanie kodu w repozytoriach zdalnych
- Wspierają pracę zespołową nad projektem
- Oferują mechanizmy pull request i merge request
- Integrują repozytoria z narzędziami CI/CD



644

Repozytoria publiczne i prywatne

- Repozytoria publiczne są dostępne dla wszystkich użytkowników
- Repozytoria prywatne są dostępne tylko dla wybranych osób
- Repozytoria publiczne są często wykorzystywane w projektach open source
- Repozytoria prywatne są stosowane w projektach komercyjnych
- Systemy hostingowe pozwalają zarządzać dostępem do repozytoriów



Kontrola wersji w środowisku akademickim i komercyjnym

- Wykorzystywana w projektach studenckich i badawczych
- Ułatwia współpracę zespołów projektowych
- Jest standardem w firmach programistycznych
- Wspiera zarządzanie rozwojem dużych systemów
- Pozwala śledzić historię rozwoju oprogramowania

646

Podsumowanie wykładu

- Systemy kontroli wersji umożliwiają zarządzanie zmianami w kodzie
- Git jest najczęściej stosowanym systemem kontroli wersji
- Repozytoria umożliwiają pracę zespołową nad projektem
- Gałęzie pozwalają rozwijać funkcjonalności równolegle
- Systemy kontroli wersji wspierają jakość i bezpieczeństwo oprogramowania

647

Dziękuję za uwagę

Wykład nr 14 Temat: Sztuczna inteligencja w tworzeniu oprogramowania

Paweł Karczmarek

Katedra Inteligencji Obliczeniowej Politechniki Lubelskiej

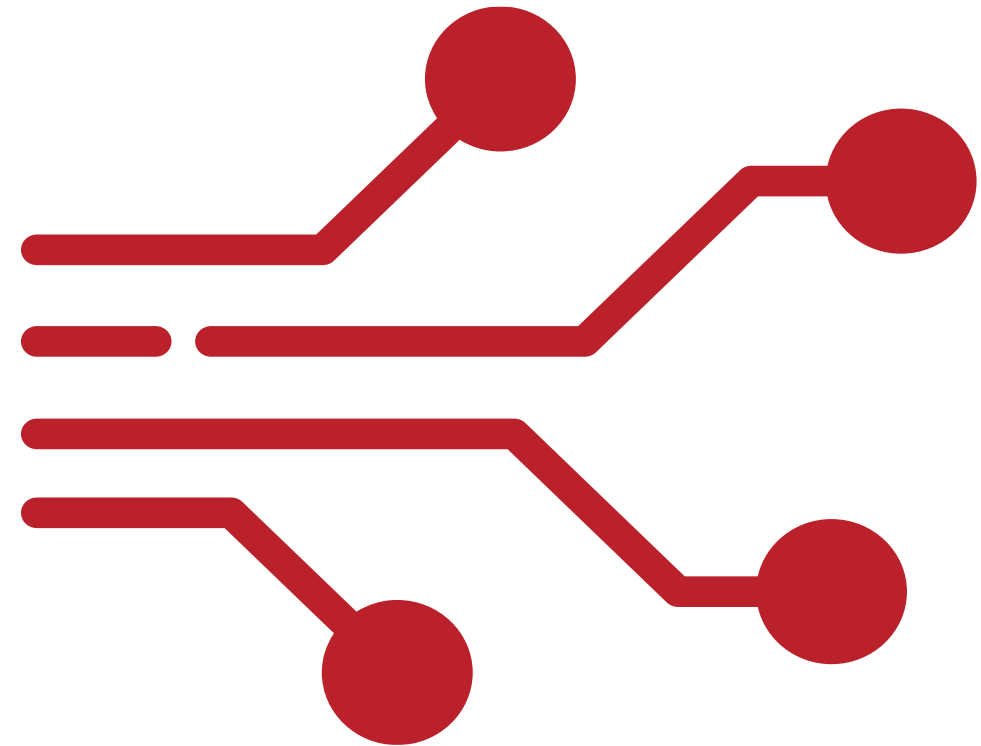
Plan prezentacji

- Wprowadzenie
- Zastosowanie



650

Wprowadzenie



651

Sztuczna inteligencja w inżynierii oprogramowania

- Wykorzystanie metod AI do wspomagania procesu tworzenia oprogramowania
- Automatyczna analiza kodu oraz generowanie fragmentów programów
- Wsparcie programistów w projektowaniu i implementacji systemów
- Wykorzystanie modeli językowych do pracy z kodem źródłowym
- Automatyzacja wybranych zadań w cyklu życia oprogramowania

652

Od automatyzacji do systemów wspomagających programistę

- Automatyzacja zadań programistycznych istnieje od wielu lat
- Kompilatory i narzędzia budowania automatyzują proces tworzenia aplikacji
- IDE wspierają programistów poprzez podpowiedzi i analizę kodu
- Systemy AI rozszerzają te możliwości o analizę semantyczną kodu
- Nowoczesne narzędzia AI mogą generować kod na podstawie opisu problemu

653

Obszary zastosowań AI w cyklu życia oprogramowania

- Analiza wymagań i przetwarzanie dokumentacji projektowej
- Wsparcie projektowania architektury systemu
- Generowanie i analiza kodu źródłowego
- Automatyczne tworzenie testów oprogramowania
- Wsparcie utrzymania i monitorowania systemów



654

Korzyści i ograniczenia wykorzystania AI

- Korzyści

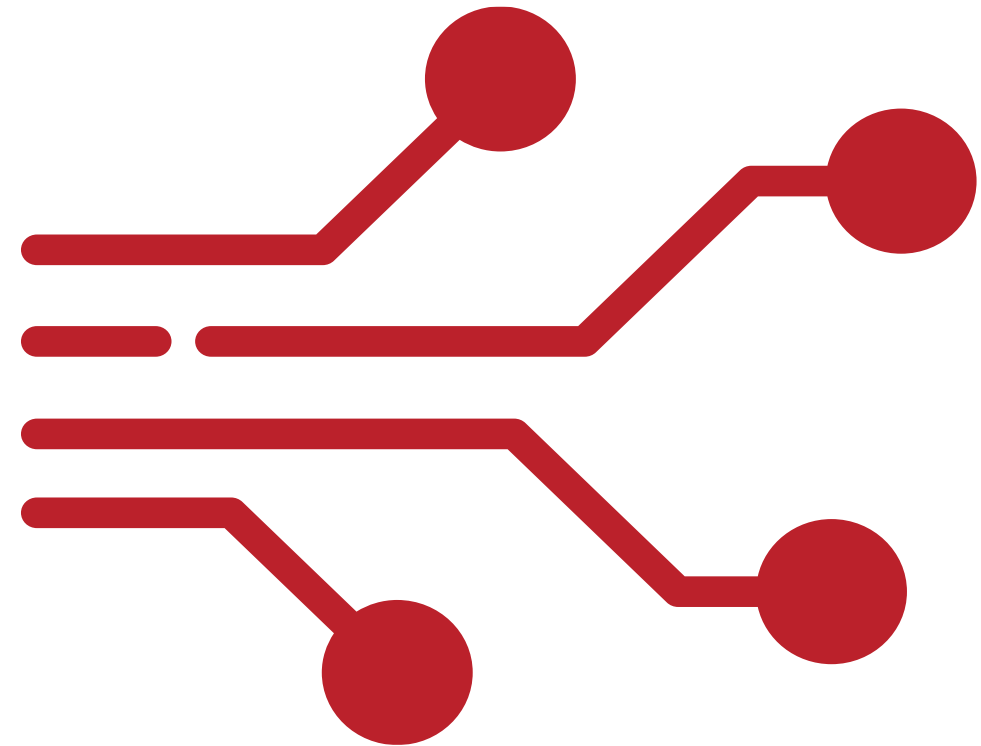
- Przyspieszenie procesu tworzenia oprogramowania
- Automatyzacja powtarzalnych zadań programistycznych
- Wsparcie w analizie i zrozumieniu istniejącego kodu
- Pomoc w nauce nowych technologii i bibliotek

- Ograniczenia

- Możliwość generowania błędnego lub niebezpiecznego kodu
- Brak pełnego zrozumienia kontekstu systemu
- Ryzyko halucynacji modeli AI
- Konieczność weryfikacji generowanych rozwiązań

655

Zastosowanie



656

AI w analizie wymagań

- Analiza dokumentów opisujących wymagania systemu
- Identyfikowanie funkcjonalności i aktorów systemu
- Wsparcie w tworzeniu specyfikacji wymagań
- Generowanie propozycji user stories
- Wykrywanie niejednoznaczności w wymaganiach



657

Generowanie user stories i przypadków użycia

- Tworzenie user stories na podstawie opisu systemu
- Identyfikowanie aktorów oraz ich celów w systemie
- Generowanie scenariuszy przypadków użycia
- Tworzenie wstępnych modeli funkcjonalnych systemu
- Wsparcie dokumentacji wymagań funkcjonalnych



Wsparcie modelowania architektury i projektowania systemu

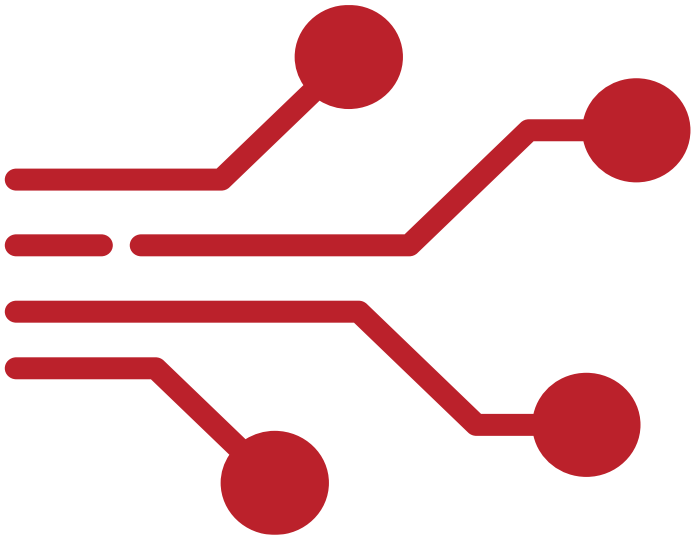
- Analiza wymagań i proponowanie architektury systemu
- Sugestie dotyczące podziału systemu na moduły
- Propozycje technologii i frameworków
- Wsparcie w tworzeniu diagramów architektury
- Analiza zależności między komponentami systemu



659

AI a tworzenie dokumentacji projektowej

- Automatyczne generowanie fragmentów dokumentacji systemu
- Tworzenie opisów modułów i funkcjonalności
- Generowanie dokumentacji API
- Podsumowywanie zmian w kodzie
- Wsparcie aktualizacji dokumentacji projektu



660

Generowanie kodu przez modele AI

- Modele AI mogą tworzyć fragmenty kodu na podstawie opisu problemu
- Generowanie funkcji, klas oraz całych modułów programu
- Wsparcie wielu języków programowania
- Możliwość szybkiego prototypowania rozwiązań
- Przyspieszenie implementacji funkcjonalności

661

Uzupełnianie kodu i podpowiedzi programistyczne

- Inteligentne podpowiedzi podczas pisania kodu
- Automatyczne uzupełnianie fragmentów funkcji i klas
- Sugestie dotyczące użycia bibliotek i API
- Wykrywanie potencjalnych błędów w kodzie
- Przyspieszenie pracy programisty w środowisku IDE



Refaktoryzacja wspierana przez AI

- Analiza struktury istniejącego kodu
- Sugestie poprawy czytelności i struktury programu
- Identyfikowanie duplikacji kodu
- Propozycje podziału kodu na moduły lub klasy
- Wsparcie w modernizacji starszych systemów



Wyjaśnianie i analiza istniejącego kodu

- Analiza działania fragmentów kodu źródłowego
- Generowanie opisów funkcji i klas
- Wyjaśnianie logiki działania programu
- Wsparcie w analizie dużych projektów programistycznych
- Pomoc w pracy z kodem legacy



AI w nauce nowych bibliotek i frameworków

- Generowanie przykładów użycia bibliotek
- Wyjaśnianie działania funkcji i klas
- Wsparcie w konfiguracji środowiska programistycznego
- Szybkie wyszukiwanie rozwiązań problemów programistycznych
- Tworzenie przykładowych implementacji



Generowanie testów jednostkowych i integracyjnych

- Automatyczne tworzenie testów jednostkowych
- Generowanie scenariuszy testów integracyjnych
- Analiza kodu w celu identyfikacji przypadków testowych
- Wsparcie w zwiększaniu pokrycia kodu testami
- Przyspieszenie procesu przygotowania testów

666

AI w wykrywaniu błędów i podatności

- Analiza kodu w celu identyfikacji błędów programistycznych
- Wykrywanie potencjalnych podatności bezpieczeństwa
- Analiza wzorców błędów w kodzie źródłowym
- Wsparcie w analizie bezpieczeństwa aplikacji
- Integracja z narzędziami analizy statycznej kodu



667

Analiza jakości kodu z użyciem AI

- Ocena czytelności i struktury kodu
- Identyfikowanie złożonych fragmentów programu
- Wykrywanie duplikacji kodu
- Analiza zgodności ze standardami programowania
- Wsparcie w utrzymaniu wysokiej jakości oprogramowania



AI a debugowanie aplikacji

- Analiza komunikatów błędów i wyjątków
- Sugestie dotyczące możliwych przyczyn błędów
- Propozycje poprawek w kodzie
- Analiza logów aplikacji
- Wsparcie w diagnozowaniu problemów systemowych



Ograniczenia AI w testowaniu i walidacji

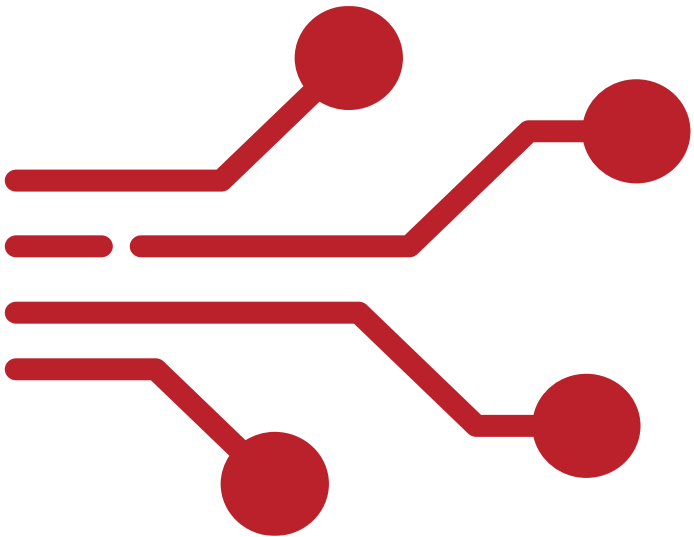
- AI nie rozumie pełnego kontekstu systemu
- Wygenerowane testy mogą nie obejmować wszystkich scenariuszy
- Ryzyko błędnych założeń w generowanych testach
- Konieczność weryfikacji testów przez programistów
- AI nie zastępuje procesu projektowania testów



670

AI w automatyzacji procesów developerskich

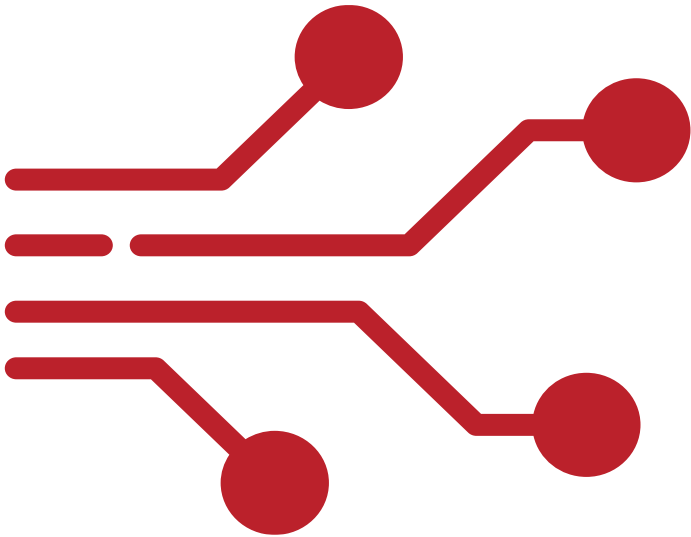
- Automatyzacja budowania i wdrażania aplikacji
- Wsparcie systemów Continuous Integration i Continuous Deployment
- Automatyczna analiza wyników testów
- Wykrywanie problemów w procesie budowania aplikacji
- Optymalizacja procesów developerskich



671

AI w monitorowaniu działania systemów

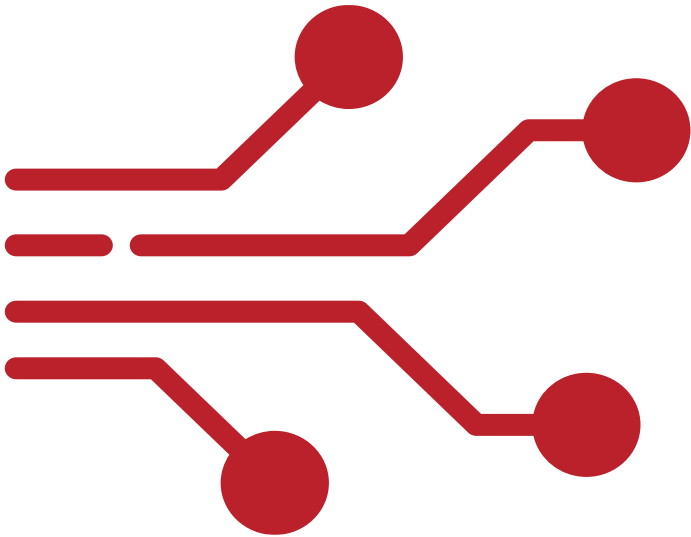
- Analiza danych z systemów monitoringu
- Wykrywanie nieprawidłowości w działaniu aplikacji
- Identyfikowanie anomalii w logach systemowych
- Wsparcie w zarządzaniu wydajnością systemu
- Automatyczne powiadamianie o potencjalnych problemach



672

Predykcja awarii i analiza logów

- Analiza logów systemowych z wykorzystaniem AI
- Identyfikowanie wzorców wskazujących na problemy systemowe
- Wykrywanie anomalii w działaniu aplikacji
- Przewidywanie potencjalnych awarii systemu
- Wsparcie w planowaniu działań utrzymaniowych



673

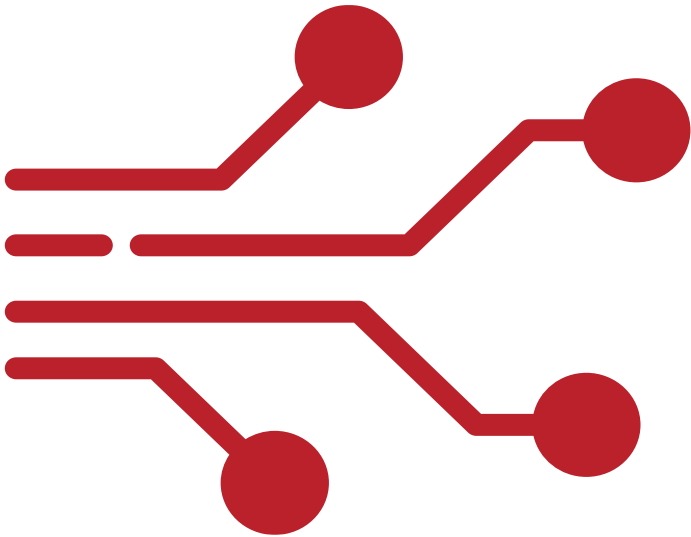
AI w utrzymaniu i konserwacji oprogramowania

- Analiza zmian w kodzie źródłowym
- Wsparcie w aktualizacji i modernizacji systemów
- Identyfikowanie przestarzałych bibliotek i zależności
- Wykrywanie potencjalnych problemów po aktualizacjach
- Wsparcie w utrzymaniu systemów legacy



Czym jest agent AI programujący

- System AI zdolny do wykonywania złożonych zadań programistycznych
- Analizuje cel zadania i planuje kolejne kroki działania
- Może czytać i modyfikować kod w repozytorium
- Wykorzystuje narzędzia programistyczne i systemy testów
- Działa w sposób częściowo autonomiczny



675

Różnica między chatbotem a agentem AI

- Chatbot AI

- Odpowiada na pytania użytkownika
- Generuje tekst lub kod na podstawie zapytania
- Nie wykonuje działań w systemie
- Działa w trybie pojedynczej odpowiedzi

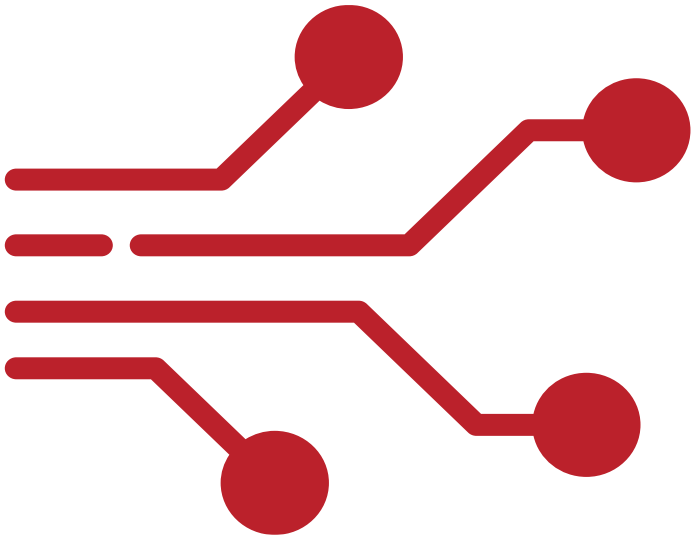
- Agent AI

- Analizuje zadanie i planuje działania
- Wykonuje wiele kroków prowadzących do rozwiązania
- Może korzystać z narzędzi i systemów zewnętrznych
- Może modyfikować kod i uruchamiać testy

676

Jak agent planuje wykonanie zadania programistycznego?

- Analiza celu zadania przekazanego przez użytkownika
- Podział problemu na mniejsze kroki
- Wybór narzędzi potrzebnych do wykonania zadania
- Wykonanie kolejnych operacji na kodzie źródłowym
- Analiza wyników i ewentualna korekta działania



677

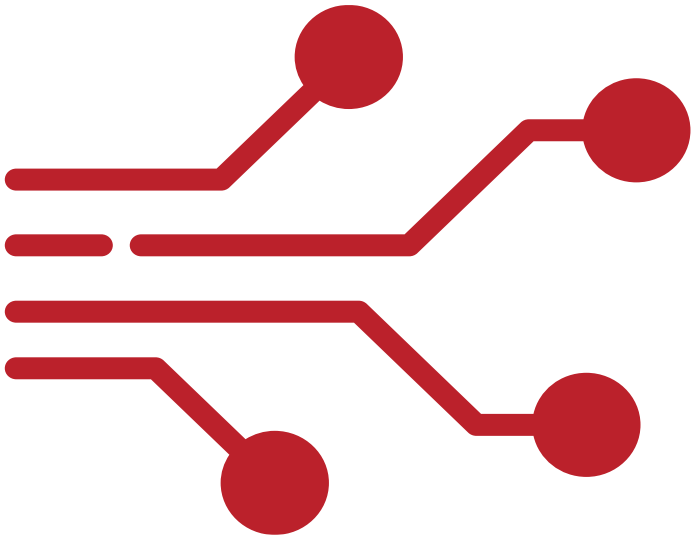
Narzędzia agenta: repozytorium, terminal, testy, dokumentacja

- Dostęp do repozytorium kodu źródłowego
- Możliwość odczytu i modyfikacji plików projektu
- Wykorzystanie terminala do uruchamiania poleceń
- Uruchamianie testów automatycznych
- Analiza dokumentacji projektu

678

Przykładowy cykl pracy agenta programistycznego

- Otrzymanie zadania od użytkownika
- Analiza kodu oraz struktury projektu
- Zaplanowanie zmian w systemie
- Wprowadzenie modyfikacji w kodzie
- Uruchomienie testów i analiza wyników
- Poprawa błędów i ponowna weryfikacja



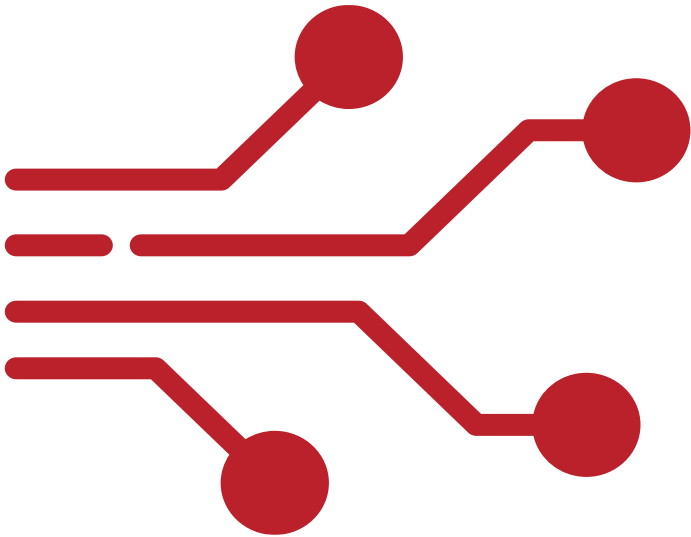
Tworzenie prostego agenta do pisania kodu

- Wykorzystanie modelu językowego (LLM)
- Zdefiniowanie celu działania agenta
- Zapewnienie dostępu do narzędzi programistycznych
- Implementacja mechanizmu planowania działań
- Weryfikacja wyników działania agenta

680

Ograniczenia agentów kodujących

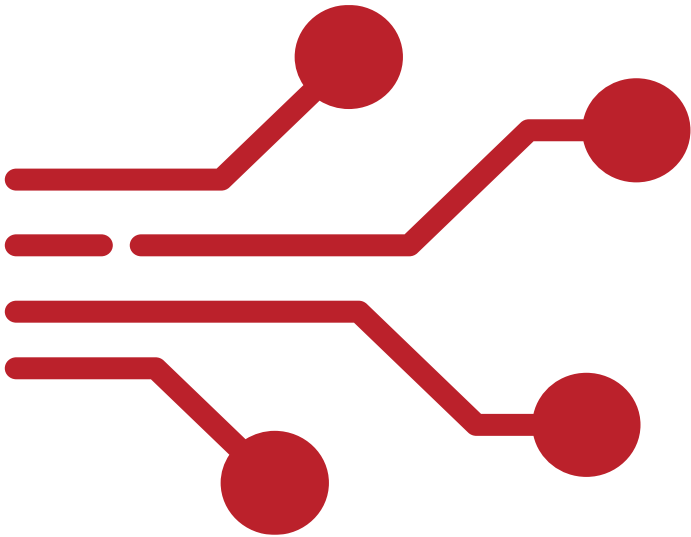
- Brak pełnego zrozumienia architektury systemu
- Możliwość generowania nieoptymalnych rozwiązań
- Ryzyko wprowadzania błędów do kodu
- Ograniczona wiedza o kontekście projektu
- Konieczność nadzoru ze strony programisty



681

Bezpieczeństwo pracy agentów programistycznych

- Ograniczenie dostępu agenta do zasobów systemu
- Kontrola zmian wprowadzanych w repozytorium
- Weryfikacja kodu generowanego przez AI
- Ochrona danych i kluczy dostępowych
- Monitorowanie działania agenta



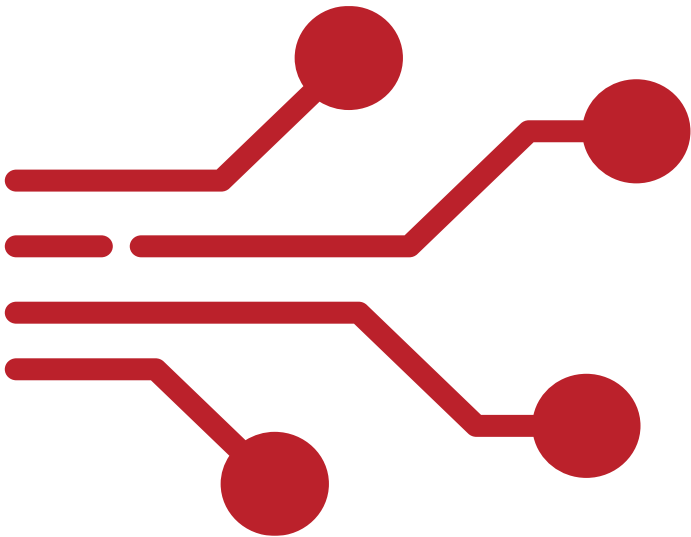
Asystenci programowania oparci na AI

- Narzędzia wspierające programistów podczas pisania kodu
- Integracja z popularnymi środowiskami IDE
- Podpowiedzi kodu oraz generowanie fragmentów programu
- Analiza błędów i sugestie poprawek
- Wsparcie wielu języków programowania

683

Chatboty i modele językowe w pracy developera

- Wykorzystanie modeli językowych do analizy kodu
- Generowanie przykładów implementacji
- Wyjaśnianie działania algorytmów
- Wsparcie w rozwiązywaniu problemów programistycznych
- Pomoc w analizie dokumentacji technicznej



IDE wspierane przez AI

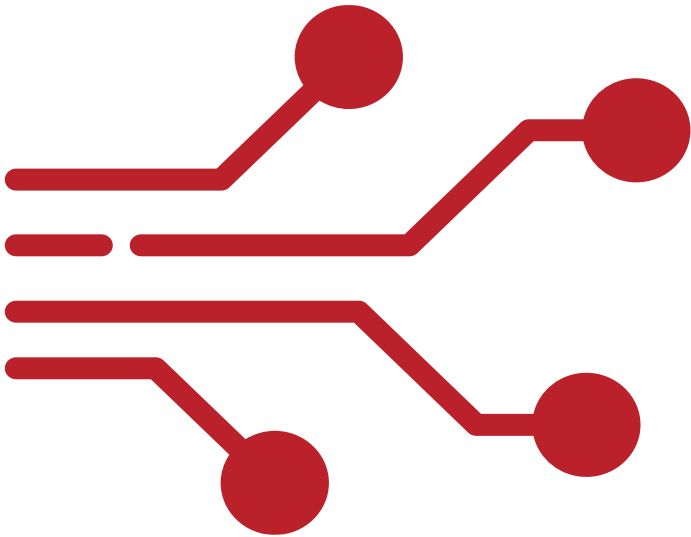
- Integracja modeli AI ze środowiskami programistycznymi
- Inteligentne podpowiedzi kodu w czasie rzeczywistym
- Automatyczna analiza błędów programistycznych
- Wsparcie w refaktoryzacji kodu
- Sugestie dotyczące optymalizacji programu



685

Przykładowy workflow pracy programisty z AI

- Analiza problemu i opis zadania
- Wygenerowanie wstępnego rozwiązania przez AI
- Modyfikacja i dostosowanie kodu przez programistę
- Generowanie testów dla nowej funkcjonalności
- Weryfikacja poprawności działania systemu



686

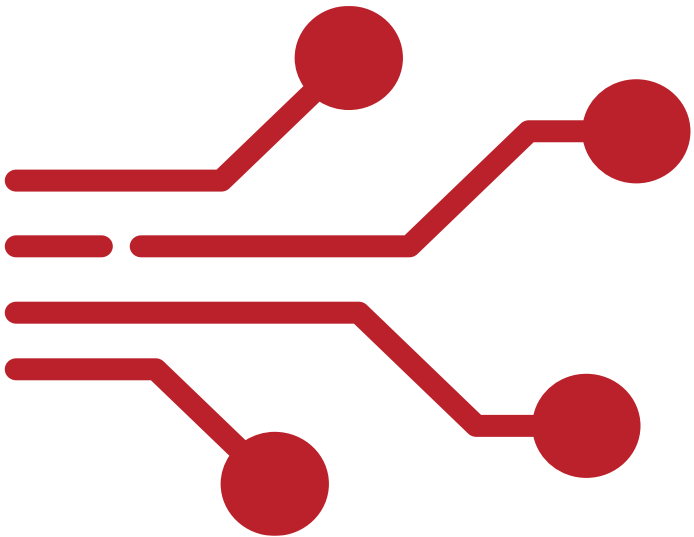
Halucynacje modeli AI

- Generowanie nieprawdziwych lub błędnych informacji
- Tworzenie kodu opartego na nieistniejących bibliotekach
- Błędne interpretowanie dokumentacji technicznej
- Tworzenie pozornie poprawnych, ale niepoprawnych rozwiązań
- Konieczność weryfikacji odpowiedzi generowanych przez AI

687

Błędy logiczne i niepoprawny kod generowany przez AI

- Kod może być poprawny składniowo, ale błędny logicznie
- AI może generować rozwiązania nieoptymalne
- Możliwe wprowadzanie podatności bezpieczeństwa
- Brak znajomości pełnego kontekstu projektu
- Konieczność przeglądu kodu przez programistów



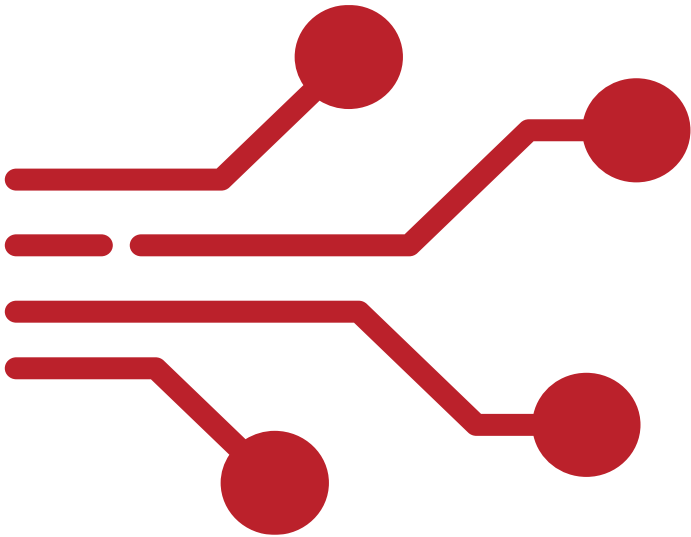
Bezpieczeństwo, prywatność i własność intelektualna

- Ryzyko ujawnienia kodu źródłowego w systemach AI
- Możliwość przetwarzania danych wrażliwych przez modele AI
- Problemy związane z prawami autorskimi do wygenerowanego kodu
- Wykorzystanie kodu z nieznanych źródeł treningowych
- Konieczność stosowania zasad bezpieczeństwa w pracy z AI

689

Dlaczego AI nie zastępuje inżyniera oprogramowania?

- AI nie posiada pełnego zrozumienia problemu biznesowego
- Nie podejmuje odpowiedzialności za architekturę systemu
- Nie rozumie wszystkich zależności w projekcie
- Wymaga nadzoru oraz weryfikacji przez programistów
- Najlepsze efekty osiąga się przy współpracy człowieka i AI



690

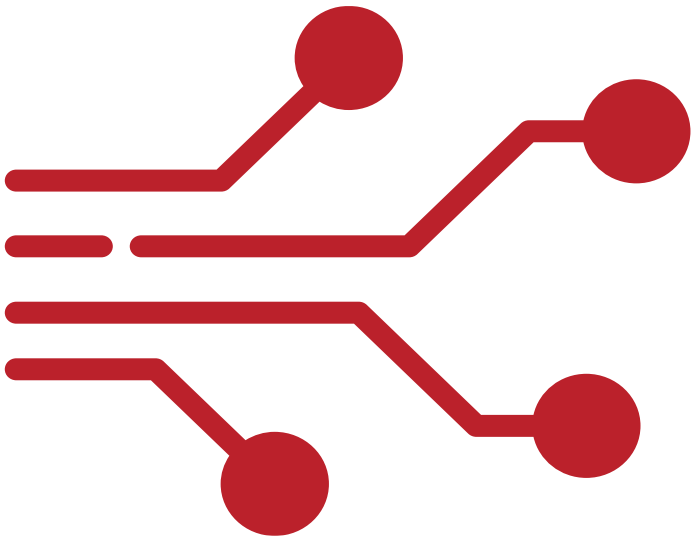
Ryzyka korzystania z publicznych modeli AI

- Możliwość przesyłania danych do zewnętrznych serwerów
- Brak kontroli nad sposobem przetwarzania danych
- Ryzyko ujawnienia kodu źródłowego lub dokumentacji
- Różnice w regulacjach prawnych między krajami
- Możliwość przechowywania zapytań użytkowników



Bezpieczne wdrożenie modelu językowego w firmie

- Uruchomienie modelu AI w infrastrukturze lokalnej (on-premise)
- Ograniczenie dostępu modelu do internetu
- Integracja z wewnętrznymi repozytoriami kodu
- Kontrola dostępu do danych projektowych
- Rejestrowanie i monitorowanie zapytań do modelu



692

Prywatny AI coding assistant w firmie

- Programista korzysta z AI bezpośrednio w środowisku IDE
- Zapytania trafiają do wewnętrznego serwera AI w firmie
- Model językowy działa w infrastrukturze organizacji
- Model analizuje lokalne repozytoria kodu i dokumentację
- Dane projektowe nie opuszczają infrastruktury firmy

Programista -> IDE (np. Visual Studio/VS Code) -> Plugin AI ->
Wewnętrzny gateway AI -> Firmowy model LLM (serwer GPU)
-> Repozytoria kodu + dokumentacja + testy

693

Zasady bezpiecznego korzystania z AI przez programistę

- Nie przysyłaj kodu źródłowego systemów produkcyjnych do publicznych modeli AI
- Nie udostępniaj kluczy API, haseł ani konfiguracji systemów
- Zawsze weryfikuj kod wygenerowany przez AI
- Sprawdzaj, czy generowany kod nie zawiera podatności bezpieczeństwa
- Traktuj AI jako narzędzie pomocnicze, a nie źródło prawdy
- Sprawdzaj dokumentację bibliotek użytych w kodzie AI
- Nie zakładaj, że wygenerowane rozwiązanie jest optymalne
- Korzystaj z AI zgodnie z polityką bezpieczeństwa organizacji
- W projektach firmowych preferuj modele uruchamiane lokalnie
- Pamiętaj, że odpowiedzialność za kod ponosi programista

694

Ile kodu generuje dziś sztuczna inteligencja?

- W wielu firmach AI wspiera znaczną część procesu programowania
- Szacuje się, że w niektórych zespołach nawet 30-42% kodu powstaje z pomocą AI
- Narzędzia AI są używane do prototypowania oraz implementacji funkcji
- Programiści coraz częściej pracują w modelu human + AI collaboration
- Kluczową rolą programisty pozostaje projektowanie i weryfikacja kodu

695

Podsumowanie

- Sztuczna inteligencja wspiera wiele etapów tworzenia oprogramowania
- AI może pomagać w analizie, programowaniu, testowaniu i utrzymaniu systemów
- Agenci AI mogą częściowo automatyzować pracę programistów
- Wykorzystanie AI wymaga weryfikacji oraz kontroli bezpieczeństwa
- Najlepsze efekty daje współpraca programisty i sztucznej inteligencji



Dziękuję za uwagę