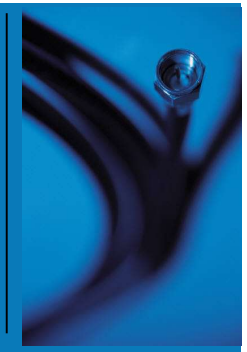
Modelowanie i analiza obiektowa

Wykład 1
„Dobre praktyki” w Inżynierii Oprogramowania
Rational Unified Process (RUP)

http://homepages.inf.ed.ac.uk/perdita/book/

Plan wykładu:



- Identyfikacja i zrozumienie problemów związanych z wytwarzaniem oprogramowania
- 6 dobrych praktyk (*Best practices*)
- RUP

Niepowodzenia

Rakieta Ariane 5.
4 czerwca 1996 – eksplozja na platformie startowej, będąca wynikiem błędów w oprogramowaniu.

Taurus.
System zawierania transakcji na Londyńskiej Giełdzie Papierów Wartościowych. Tworzenie oprogramowania 5 lat. Koszt 75 mln funtów. Straty klientów giełdy – 450 mln funtów.

System obsługi lotniska w Denver.
Składał się z ponad 300 komputerów. Początkowy koszt - 200 mln dolarów. Budżet przekroczony o ponad 50%. Wstrzymano o kilka miesięcy otwarcie lotniska.

Therac-25.
W latach 1985-87 25 osób ucierpiało z powodu otrzymania nadmiernych dawek promieniowania spowodowanych błędnym działaniem oprogramowania systemu do radioterapii. 3 osoby zmarły.

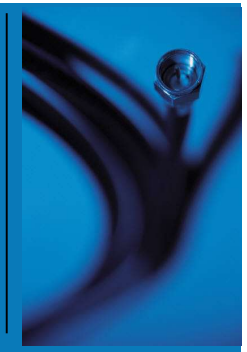
Niepowodzenia

Informatyzacja w służbie zdrowia zakończyła się wreszcie sukcesem. Mamy wreszcie e-recepty!

Jednak według zapewnień polityków i firm wdrażających system, projekt miał być gotowy pod koniec 2015 roku, albo na początku 2016. Jednak już wtedy w środowisku medycznym nie brakowało opinii, które nie dawały temu wiary. - Nieoficjalnie mówiono, że dostawcy mieli oddać platformę do testowania w sierpniu, ale według moich informacji poprosili o czas do końca roku - mówił Krzysztof Nyczaj, ekspert Izby Medycyna Polska i konsultant GUS. I dodał, że z konieczną fazą testów może to oznaczać ostateczną gotowość P1 dopiero w połowie przyszłego roku.

Niepowodzeniem jest tutaj bardzo duże opóźnienie w realizacji projektu.

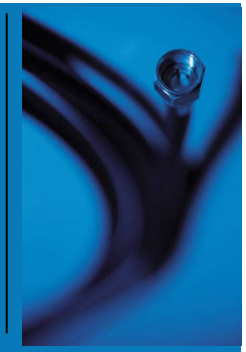
Statystyka



Na podstawie przeprowadzonych badań W. Wayt Gibs sformułował trzy wnioski:

- Przeciętne opracowanie dużego projektu pochłania 50% więcej czasu, niż zakładano.
- Trzy czwarte (!!!) dużych projektów to operacyjne porażki.
- Jedna czwarta dużych projektów zostaje anulowana.

Problemy związane z wytwarzaniem oprogramowania



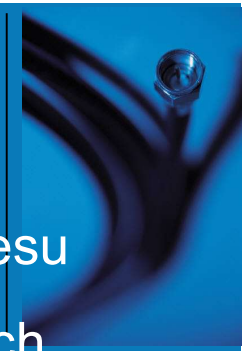
- Niezrozumienie potrzeb użytkowników (język, terminologia)
- Nieumiejętne reagowanie na zmiany w wymaganiach
- Moduły, które do siebie nie pasują
- Trudności z konserwacją systemu
- Późne wykrywanie poważnych błędów
- Zła jakość kodu (kod spaghetti)
- Słaba wydajność
- Brak koordynacji w pracy zespołowej
- Problemy z wytwarzaniem i wersjonowaniem aplikacji systemu.

Problemy i przyczyny

Przyczyną wymienionych problemów jest **niewłaściwe zarządzanie wymaganiami**. Prowadzi to do sytuacji, w której zespół projektowy nie jest w stanie dostarczyć odpowiedniego produktu w założonym czasie.

Stan taki wynika z subiektywnej oceny postępów projektu bazującej na subiektywnych metodach szacowania zaawansowania pracy.
Powoduje powstanie tzw. zjawiska „90% zrobione – 90% do zrobienia”.
Postępy pracy oszacowane na 90% ale do dokończenia potrzeba jeszcze 90% całości zasobów.

Recepta → Dobre praktyki

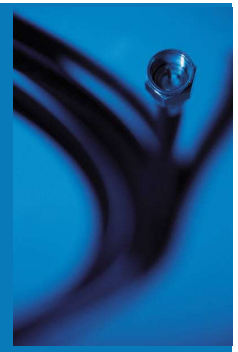


- Dobre praktyki (ang. *best practices*) są zbiorem podejść do procesu wytwarzania oprogramowania, które sprawdziły się w praktyce. Ich zaletą jest to, że jeżeli stosujemy je jednocześnie to eliminujemy najczęstsze przyczyny problemów związanych z procesem wytwarzania oprogramowania.
- Nazywane są dobrymi praktykami nie dlatego, że jesteśmy w jakiś sposób w stanie precyzyjnie zmierzyć ich jakość, ale raczej dlatego, że zaobserwowano, że są powszechnie wykorzystywane przez firmy, które odnoszą sukcesy rynkowe.
- **Można powiedzieć, że „dobre praktyki” to wynik tysięcy projektów, wypadkowa doświadczeń profesjonalistów – praktyków.**

Dobre praktyki

1. Zarządzaj wymaganiami
2. Modeluj wizualnie
3. Buduj przyrostowo
4. Wykorzystuj architekturę komponentową
5. Stale weryfikuj jakość
6. Zarządzaj zmianami

Od objawów do dobrych praktyk



Objawy	Przyczyny	Dobre praktyki
Niezrozumienie potrzeb	Brak zarządzania wymaganiami	Buduj przyrostowo
Złe reakcje na zmiany	Problemy z komunikacją	Zarządzaj zmianami, modeluj wizualnie
Problemy z integracją modułów	Krucha architektura	Zarządzaj wymaganiami
Trudności z konserwacją	Nie wykryte niespójności	Wykorzystuj architekturę komponentową
Późne wykrywanie błędów	Niewystarczające testowanie	Stale weryfikuj jakość
Zła jakość kodu	Subiektywne oceny postępów,	Modeluj wizualnie (UML), Stale weryfikuj jakość
Słaba wydajność	Model kaskadowy	Wykorzystaj model komponentowy
Brak koordynacji	Niekontrolowana propagacja zmian	Zarządzaj zmianami
Problemy z wersjonowaniem	Niewystarczające wsparcie narzędziowe	Zarządzaj zmianami

Reguły

**Zapobiegaj i walcz z przyczynami,
a wyeliminujesz objawy.**

Wyeliminuj objawy, a będziesz miał większą szansę na to, że twoje oprogramowanie będzie lepszej jakości.

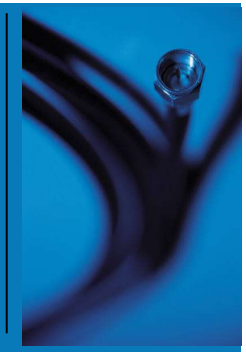
Praktyka 1: Zarządzaj wymaganiami

- Jednymi z podstawowych czynników porażki projektu programistycznego są:
  - Niepoprawne definiowanie wymagań (nieznajomość języka obcego, nieznajomość słownictwa branżowego)
  - Brak zarządzania wymaganiami w czasie trwania projektu (wszelkie zmiany wymagań NATYCHMIAST powinny być odnotowane w specyfikacji wymagań a członkowie zespołu o tym fakcie poinformowani – częstość organizowania zebrań).

Co to jest „wymaganie”?

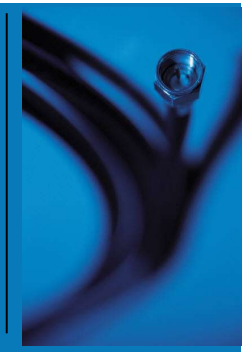
Wymaganie – warunek stawiany przez zamawiającego oprogramowanie lub własność jaką musi posiadać system.

Aspekty zarządzania wymaganiami



- Zrozumienie potrzeb użytkowników
- Analiza problemu
- Zdefiniowanie systemu
- Określenie zasięgu działania systemu
- Udoskonalanie definicji budowanego systemu
- Sprawdzanie (testowanie) działania właściwego systemu

Zarządzanie wymaganiami



Upewnij się, że:

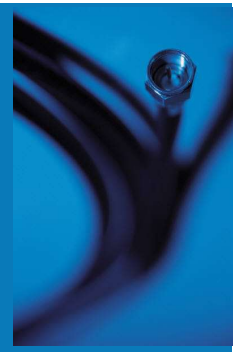
- Rozwiązujesz właściwy problem,
- Tworzysz właściwy system

Pewność tę można uzyskać poprzez systematyczne podejście do:

- Pozyskiwania wymagań
- Organizowania i dokumentowania zdobytej wiedzy
- Zarządzania tą wiedzą

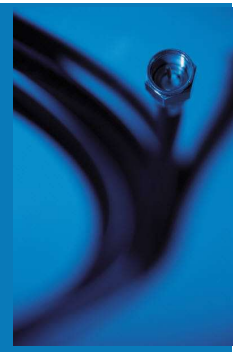
Należy pamiętać, że niektóre z wymagań mogą się zmieniać w trakcie trwania projektu.

Zarządzanie wymaganiami i „traceability” (identyfikowalność)



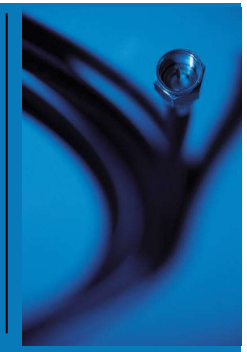
- Zarządzanie wymaganiami dotyczy procesu translacji wymagań klientów w **zbiór kluczowych potrzeb i cech systemu.**
- Następnie tenże zbiór (cech i potrzeb) jest przekształcany w **specyfikację funkcjonalnych i niefunkcjonalnych wymagań.**
- Ta szczegółowa specyfikacja jest następnie przekształcana w **projekt**, **procedury testowe** i **dokumentację użytkownika**.

Zarządzanie wymaganiami i „traceability” (identyfikowalność)



- „Traceability” polega na:
  - Oszacowaniu wpływu zmiany w wymaganiach na projekt.
  - Oszacowaniu wpływu jaki na wymagania będzie miał nie zaliczony test (jeżeli test nie został zdany wymagania mogą nie być spełnione)
  - Wyznaczanie ram projektu
  - Zweryfikowanie czy wszystkie wymagania w implementowanym systemie zostały spełnione
  - Zweryfikowanie czy system realizuje założone cele.
  - Zarządzanie zmianami.

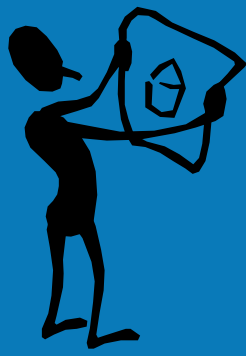
Praktyka 2: Modeluj wizualnie



- Model jest uproszczoną wersją rzeczywistości (obrazem) zawierającą te jej elementy, które należą do pewnego poziomu abstrakcji (są istotne z danego punktu widzenia).
- Modele pozwalają na opisanie systemu w sytuacji, gdy nie jesteśmy w stanie pojąć jego złożoności.

Dlaczego modelujemy wizualnie?

- Ułatwienie komunikacji między członkami zespołu
  - Jeden wspólny język (UML)
  - Automatyczne narzędzia
  - Ukrywanie bądź eksponowanie szczegółów
- Zarządzanie spójnością pomiędzy różnymi artefaktami* systemu.
- Zwiększanie poziomu abstrakcji



*Artefakty: wymaganie, projekt, implementacja (kod źródłowy), scenariusze testów, ...

Dlaczego modelujemy wizualnie? c.d.

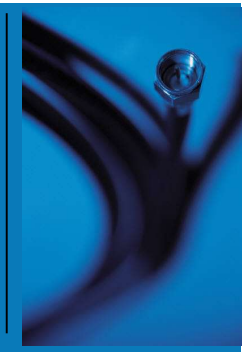
Modelowanie wizualne wspomaga zespół projektantów w walce ze złożonością oprogramowania

Ogólna zasada:
Modelowanie jest tym ważniejsze im projekt jest bardziej złożony i wymaga większych nakładów

Modele dzielą się na dwa rodzaje:

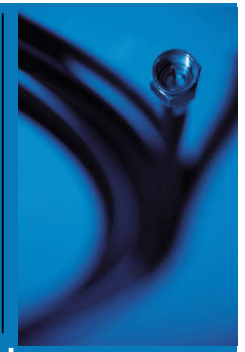
- Model strukturalny – kładący uwagę na organizację systemu
- Model dynamiczny – pokazujący działanie systemu

Modelowanie przy pomocy UML

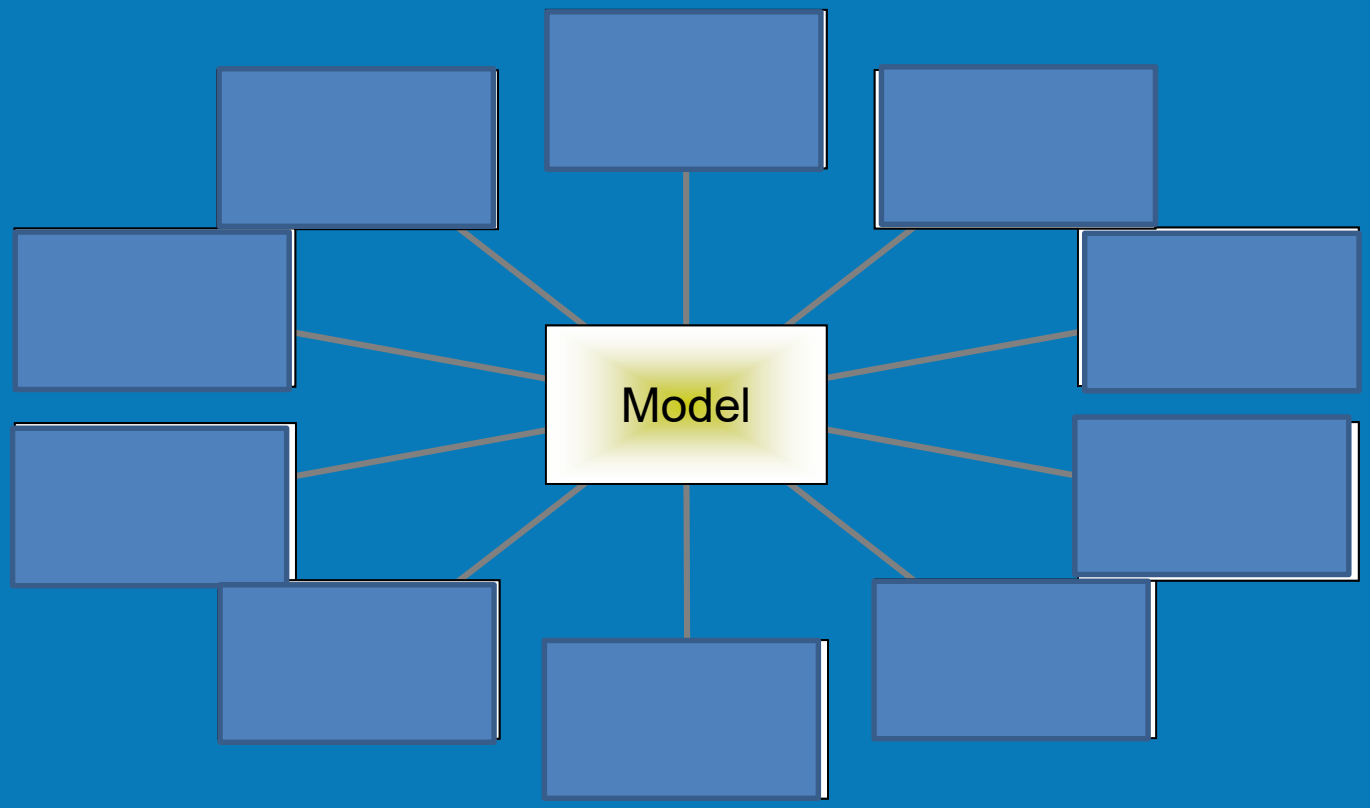


- Wspólny język graficzny
- Wsparcie narzędziowe
- Standard przemysłowy
- Spójny, umożliwiający jednoznaczną komunikację
- Wiele punktów widzenia na system i jego działanie

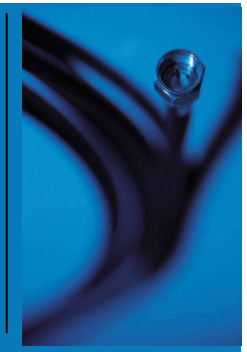
Wiele punktów widzenia



Tworząc system potrzebujemy spojrzenia na jego własności z różnych punktów widzenia. W języku UML istnieją różne typy diagramów umożliwiające opis projektowanego systemu z różnych perspektyw:

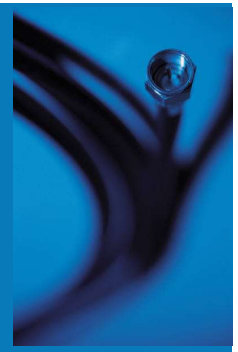


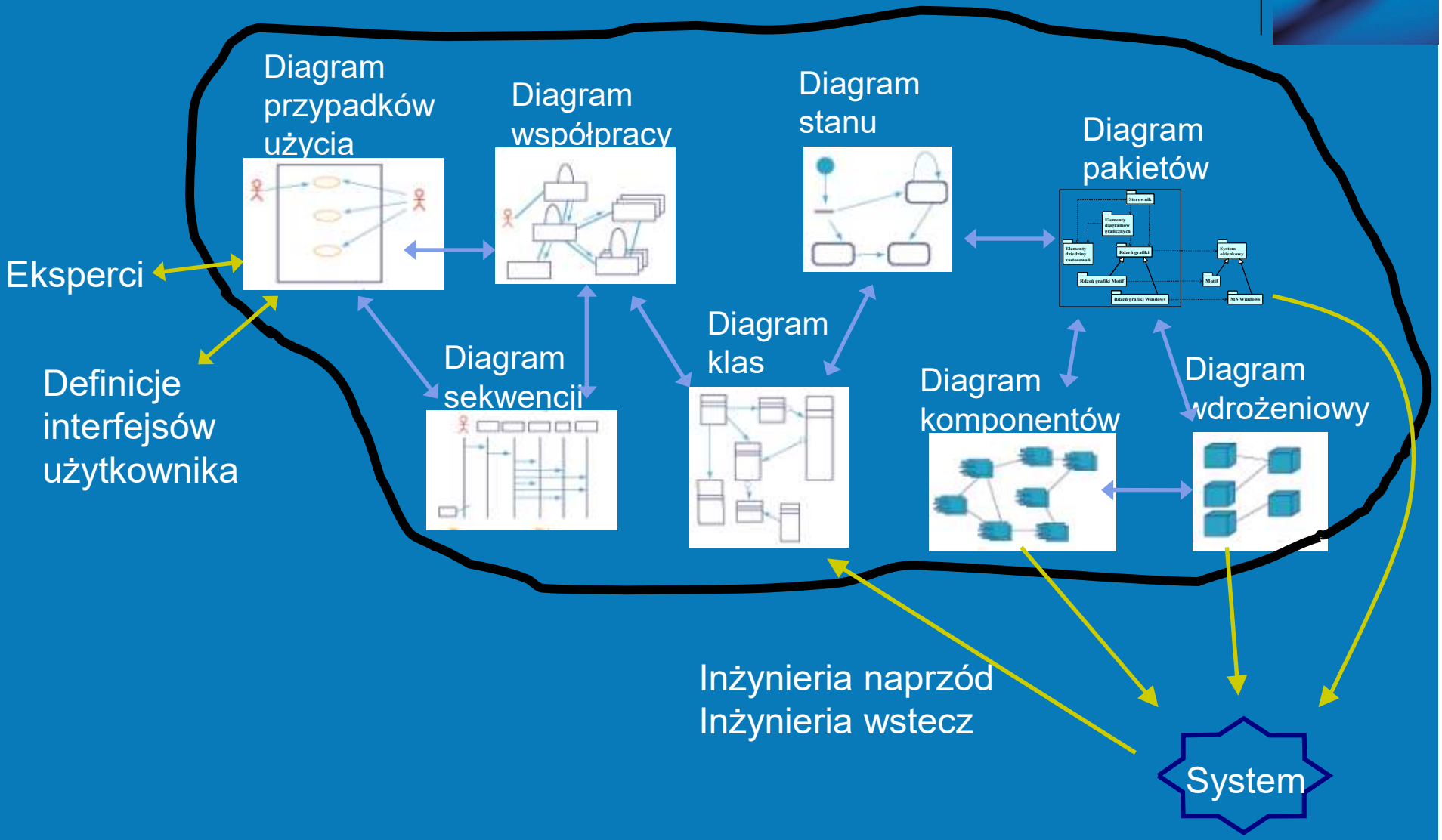
Rodzaje diagramów UML



- **Przypadków użycia**: model interakcji użytkownika z systemem
- **Pakietów: grupowanie modeli w logicznie połączone zbiory**
- **Klas**: opis logicznej struktury systemu
- **Obiektów**: opis obiektów występujących w systemie oraz powiązań pomiędzy nimi
- **Stanu**: opis zachowania systemu
- **Komponentów**: opis fizycznej struktury oprogramowania
- **Wdrożeniowe**: opis sposobu mapowania oprogramowania na konfigurację sprzętową
- **Interakcji** (sekwencji, współpracy): ilustracja zachowania systemu w czasie
- **Aktywności**: opis przepływu sekwencji zdarzeń

UML jako narzędzie





Podsumowanie modelowania wizualnego

- Dzięki modelowaniu wizualnemu projektowana architektura systemu staje się lepiej rozumiana. Łatwiej jest zrozumieć skutki podjętych decyzji.
- Dodatkową zaletą udostępnianą przez automatyczne narzędzia jest możliwość dokonywania inżynierii naprzód i inżynierii wstecz:
  - inżynieria naprzód polega na automatycznym przekształcaniu modelu do kodu źródłowego,
  - inżynieria wstecz jest procesem odwrotnym. Umożliwia odzyskiwanie informacji o szczegółach projektu systemu na podstawie kodu źródłowego (zazwyczaj jest to diagram klas).

Latest news.

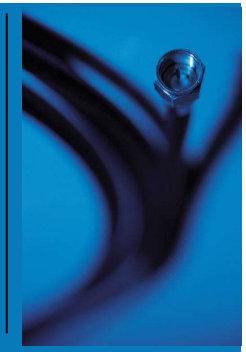
Za chwilę minie rok od czasu, kiedy samolot Boeing 737 MAX, należący do indonezyjskich linii lotniczych Lion Air, runął ze 189 ludźmi na pokładzie w rejonach wyspy Jawa. Z kolei 157 ofiar śmiertelnych pochłonęła katastrofa samolotu Ethiopian Airlines — w marcu 2019 r. W tym przypadku też w roli głównej występował bodaj najpopularniejszy samolot na świecie: Boeing 737 MAX.

Testy nowego oprogramowania, czyli drugie życie dla Boeinga
W listopadzie wyselekcjonowana grupa pilotów (nie tylko z amerykańskich linii lotniczych) rozpocznie testy symulatorów nowego oprogramowania dla Boeinga 737 MAX. Bedą przeprowadzone w siedzibie lotniczego giganta — w Seattle. Wszystkiemu bacznie mają przyglądać się przedstawiciele FAA. Wtedy ma się okazać, czy pilotując 737 MAX, cały czas natrafia się na kłopoty, które były przyczyną dwóch ostatnich katastrof.

Wielu ekspertów uważa, że w przypadku tych dwóch katastrof najbardziej zawiodły systemy zautomatyzowane. Dlatego, jak donosi CNN, piloci w symulatorze doświadczą scenariuszy i awarii sterowania lotem, które odzwierciedlają trudności, jakie przeżywali piloci w dwóch śmiertelnych wypadkach 737 MAX.

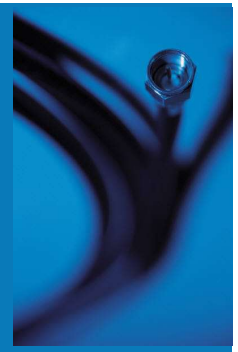
Wyniki testów z pewnością będą bardzo ważne. Tak dla samego Boeinga, jak i całej branży. Jednak nawet ich pozytywne zakończenie nie musi oznaczać globalnego powrotu na lotniska samolotu Boeing 737 MAX. Na myśl o takim scenariuszu coraz bardziej nosem bowiem kręcą Australijczycy. Ciągle wszak jest żywa obawa, że Amerykanie niezbyt dokładnie patrzą Boeingowi na ręce.

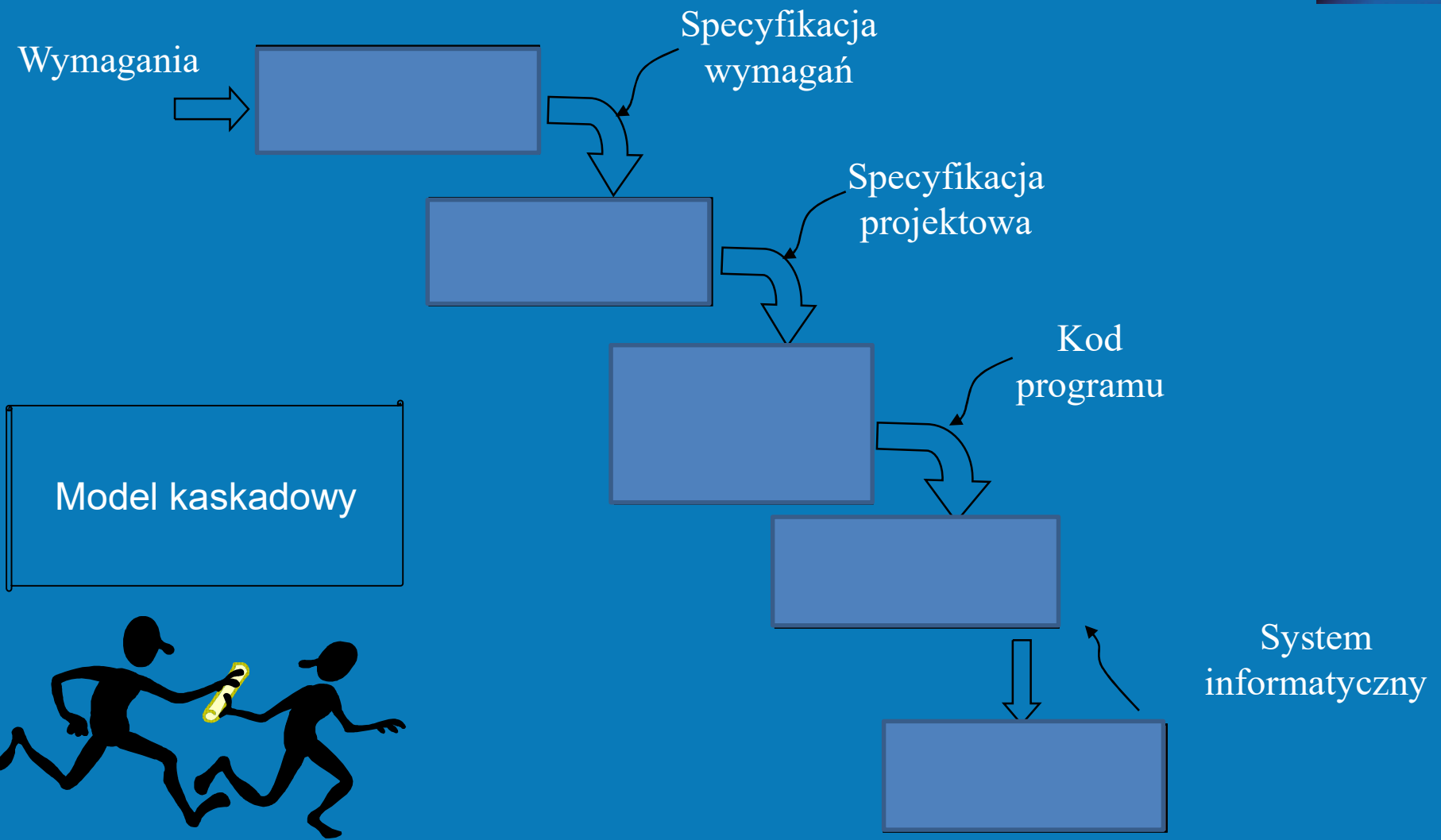
Praktyka 3: Buduj przyrostowo
(ang. Develop Iteratively)



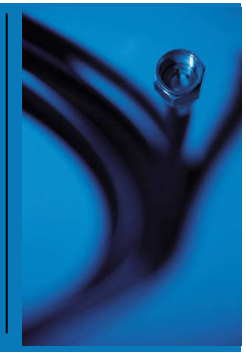
- Dostarczanie systemu o wzrastającej funkcjonalności w serii kolejnych wydań. Każde wydanie jest tworzone w ustalonym okresie czasu (iteracja).
- Każda iteracja skupia się na zdefiniowaniu, przeanalizowaniu, zaprojektowaniu, zaimplementowaniu i przetestowaniu określonego zbioru wymagań.

Model kaskadowy



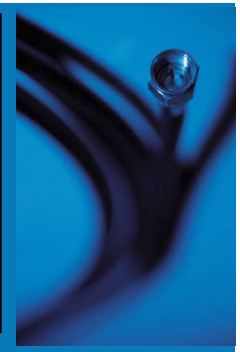


Model kaskadowy



- Model kaskadowy jest z koncepcyjnego punktu widzenia wygodny do stosowania (system jest dostarczany jeden raz).
- Zagrożeniem w takim podejściu jest odkładanie w czasie ryzyka niepowodzenia (wystąpienia błędów) do momentu, w którym ich naprawa z wcześniejszych faz jest już bardzo kosztowna.

Model iteracyjny

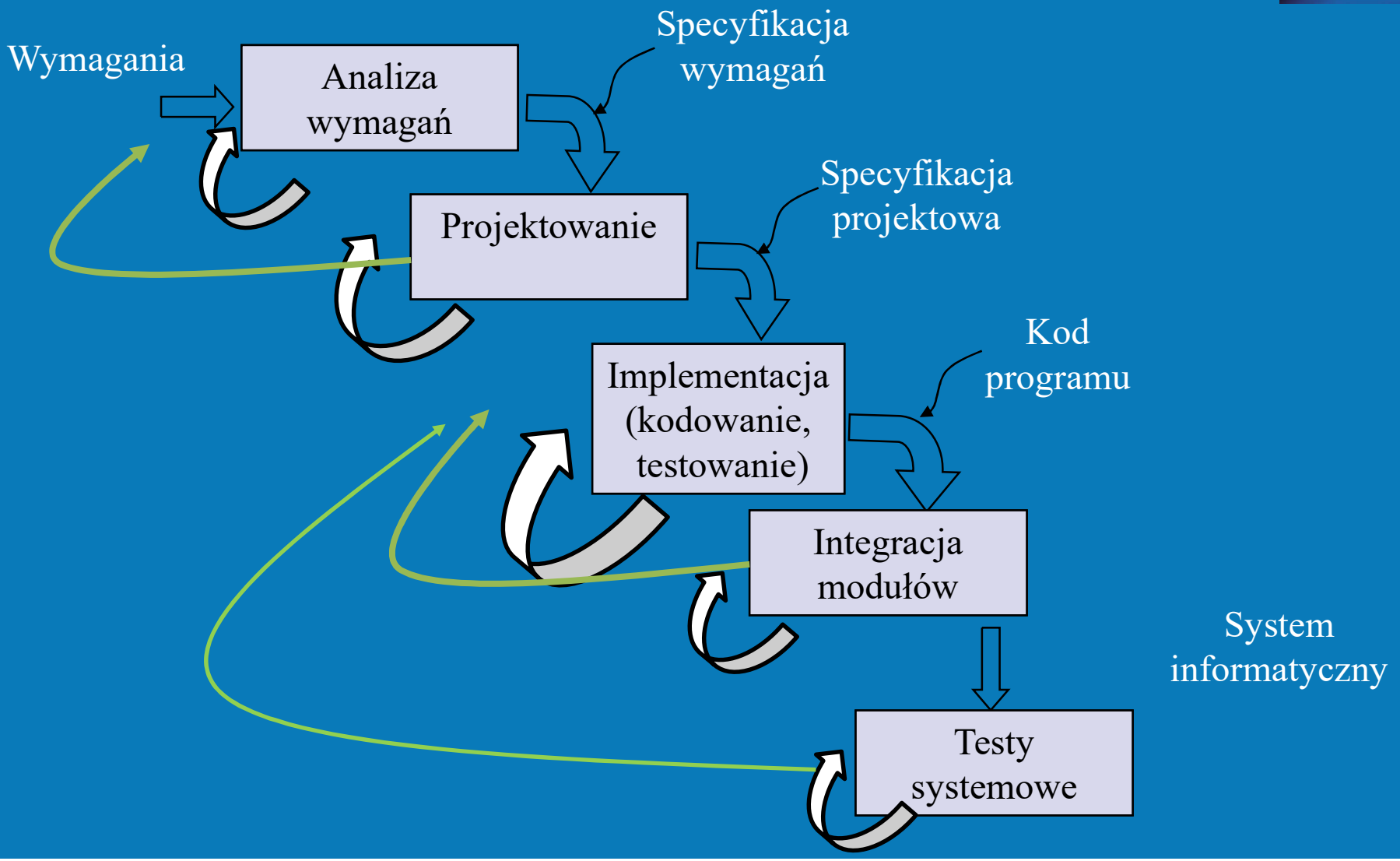


Każda iteracja daje w efekcie poprawnie działające oprogramowanie!

Model kaskadowo - iteracyjny

- W procesie iteracyjnym kroki modelu kaskadowego są wykonywane iteracyjnie. Zamiast tworzyć od razu system w całości, wybierany jest podzbiór funkcjonalności, implementowany, potem wybierany kolejny itd.
- **Pierwsza iteracja jest oparta na elementach powiązanych z ryzykiem o najwyższym priorytecie.**

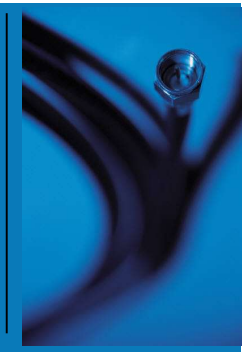
Model kaskadowo-iteracyjny

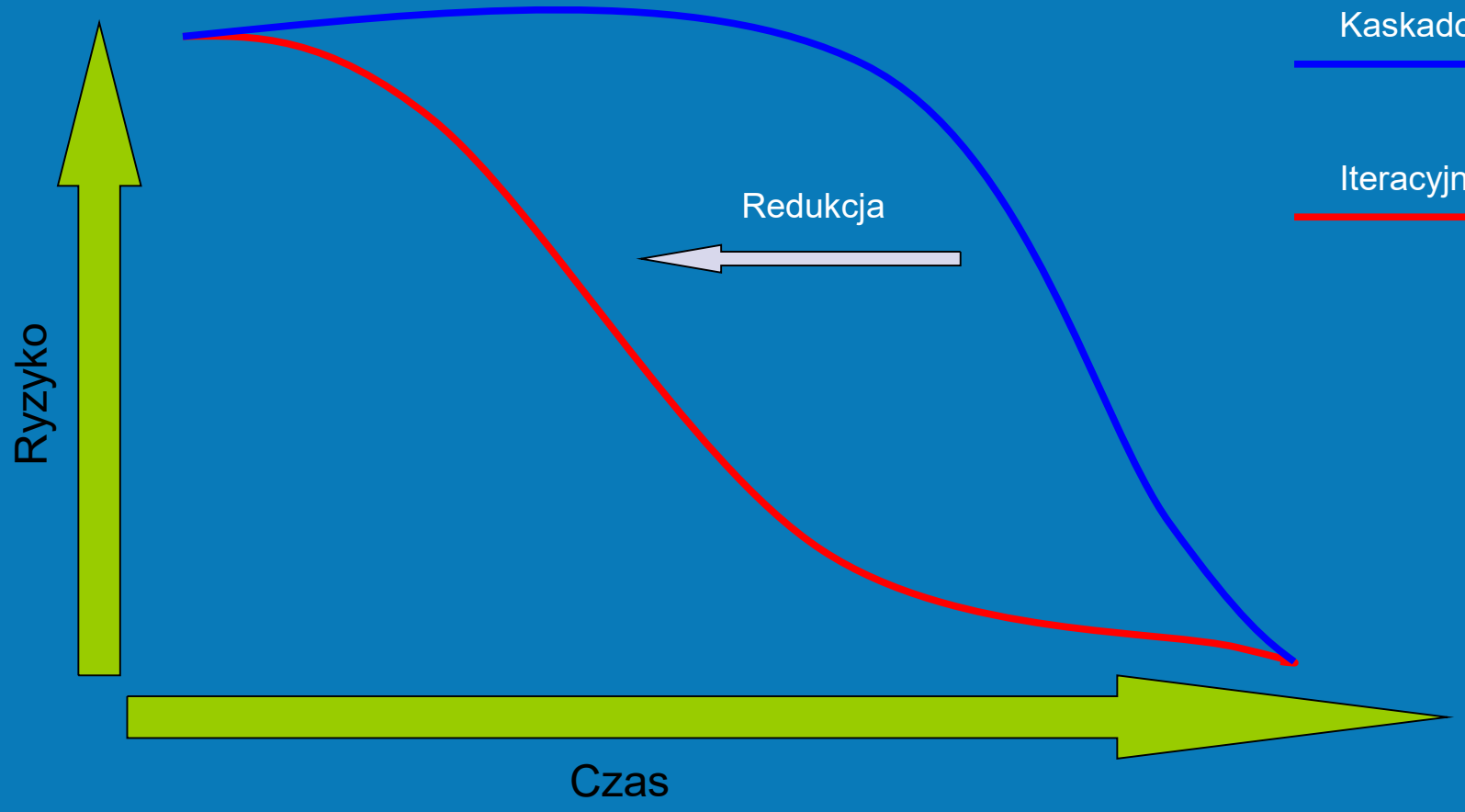


Model kaskadowo - iteracyjny

- **Zalety:**
  - **Największe ryzyko występuje i jest eliminowane we wczesnej fazie (być może jeszcze przed większymi inwestycjami)**
  - **Bardzo szybko mamy informacje zwrotne od klienta (już po pierwszej iteracji)**
  - **Integracja i testowanie są procesami ciągłymi**
  - **Skupianie się na krótkoterminowych celach (kamienie milowe projektu – ang. *milestones*)**
  - **Umożliwia wdrażanie niepełnych implementacji systemu.**

Ryzyko





Zalety:

- Model kaskadowo - iteracyjny umożliwia weryfikację architektury w fazie integracji każdej z iteracji.
- Słabości projektu mogą być wykryte wcześniej.
- Ciągła integracja przez cały czas trwania projektu zastępuje pojedynczą integracje na końcu projektu.
- Lepsze jest zarządzanie jakością ponieważ pewne niefunkcjonalne elementy architektury takie jak wydajność czy tolerancja na błędy są wykrywane wcześnie.

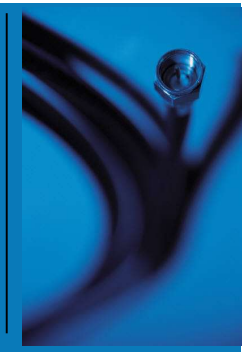
Praktyka wykonywania projektów

Badania wykazują, że niewielka liczba projektów jest wykonywana na czas i w granicach założonego budżetu:

- 16% projektów na czas i w określonym budżecie
- 53% projektów przekraczających terminarz i budżet
- 31% projektów zakończonych porażką

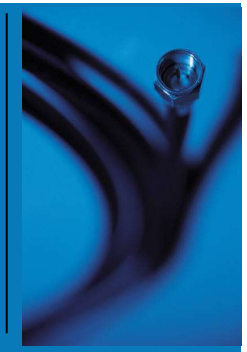
źródło: www.standaishgroup.com

Praktyka 4: Wykorzystuj architekturę komponentową



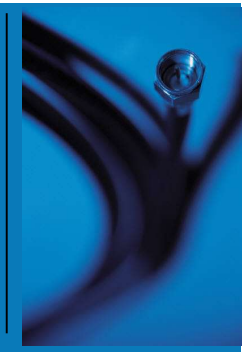
Według SEI (*Software Engineering Institute*, http://www.sei.cmu.edu/) wykorzystanie odpowiedniej architektury może zwiększyć produktywność o 10%.

Architektura



- Definiuje sposób, w jaki system będzie budowany.
- Ma ona ogromny wpływ na możliwości ewolucji systemu oraz jego wydajność.

Architektura



- **Najważniejszą pożądaną cechą architektury jest elastyczność czyli łatwość późniejszych zmian**. Aby wbudować w system taką elastyczność architekt musi przewidzieć nie tylko **ewolucję wymagań** ale również **ewolucję technologii**. Podstawowymi technikami, które umożliwiają mu osiągnięcie tego celu są:
  - abstrakcja,
  - hermetyzacja,
  - obiektowo zorientowana analiza i projektowanie.
- Celem jest aplikacja, którą łatwo się konserwuje i łatwo rozbudowuje.

Co to jest „komponent”?

Niezależnie wytworzony, skompilowany (z ukrytymi szczegółami implementacyjnymi) ***moduł programowy***, udostępniający swą funkcjonalność za pomocą jednoznacznie zdefiniowanego ***interfejsu***, zdolny do współdziałania z większą całością (systemem) oraz innymi komponentami.

Cechy komponentu:

- komponent jest jednostką niezależnego przetwarzania,
- komponent jest jednostką niezależnej asemblacji,
- komponent nie ma żadnego trwałego stanu.

Komponenty występują m.in. w Delphi, Visual Basic, KDE, GNOME, technologii .NET, a także w CMS - Joomla! [Wikipedia]

Co to jest „komponent”?

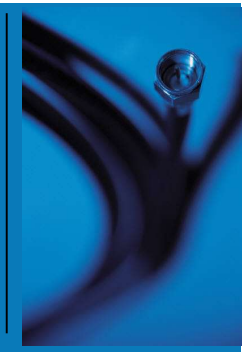
Definicja komponentu podana przez C. Szyperskiego:

- komponent jest podstawowym elementem, z którego budowane jest oprogramowanie,
- posiada opisane deklaratywnie interfejsy, a wszystkie wymagane przez niego zależności są podane wprost. Dzięki temu możliwe jest przekazanie go do wdrożenia osobie, która nie jest jego twórcą.

Komponenty na poziomie implementacyjnym są zwykle rozszerzeniem obiektów (choć istnieją także technologie komponentowe zbudowane w oparciu o języki strukturalne), dlatego komponent jest zbudowany z jednej lub kilku klas.

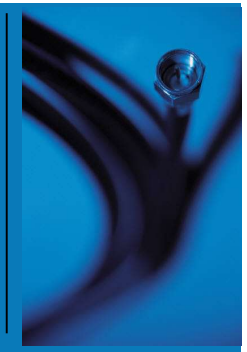
Dzięki deklaracjom oferowanych interfejsów i żądanych zależności komponent może być traktowany jako zamknięta całość, która oferuje usługi i żąda ich od zewnętrznych komponentów, jednocześnie ukrywając swoją strukturę wewnętrzną i nie wnikając w budowę innych. To czyni z komponentu jednostkę bardziej abstrakcyjną niż obiekt.

Architektura komponentowa



- **Elastyczność**
  - Zaspokojenie obecnych i przyszłych wymagań
  - Zwiększenie możliwości rozbudowy systemu
  - Umożliwienie ponownego użycia
  - Hermetyzacja zależności pomiędzy elementami systemu
- **Modularność**
  - Powtórne użycie elementów (komponentów)
  - Możliwość zastosowania istniejących komercyjnych komponentów
  - Iteracyjne ewoluowanie systemu

Obiekty a komponenty



Programowanie obiektowe

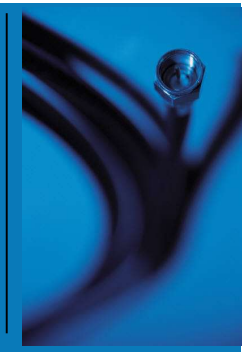
- polimorfizm
- późne wiązanie wywołań
- częściowa hermetyzacja
- dziedziczenie klas

Programowanie komponentowe

- konfiguracja wdrożenia
- późne wiązanie wywołań i ładowanie kodu, pełna hermetyzacja
- dziedziczenie interfejsów
- powtórne użycie na poziomie binarnym

Dobrze zaprojektowany obiekt (lub ich grupa) w wielu przypadkach może być użyty jako komponent. Jednak stopień abstrakcji i hermetyzacji obiektu zależy w dużej mierze od projektanta.

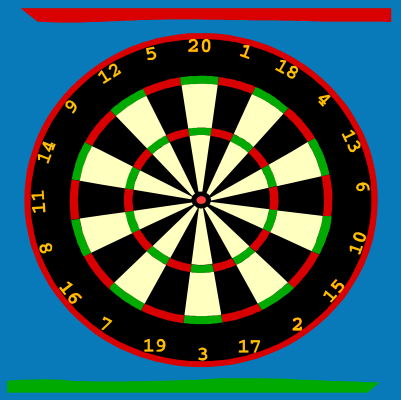
Własności komponentu

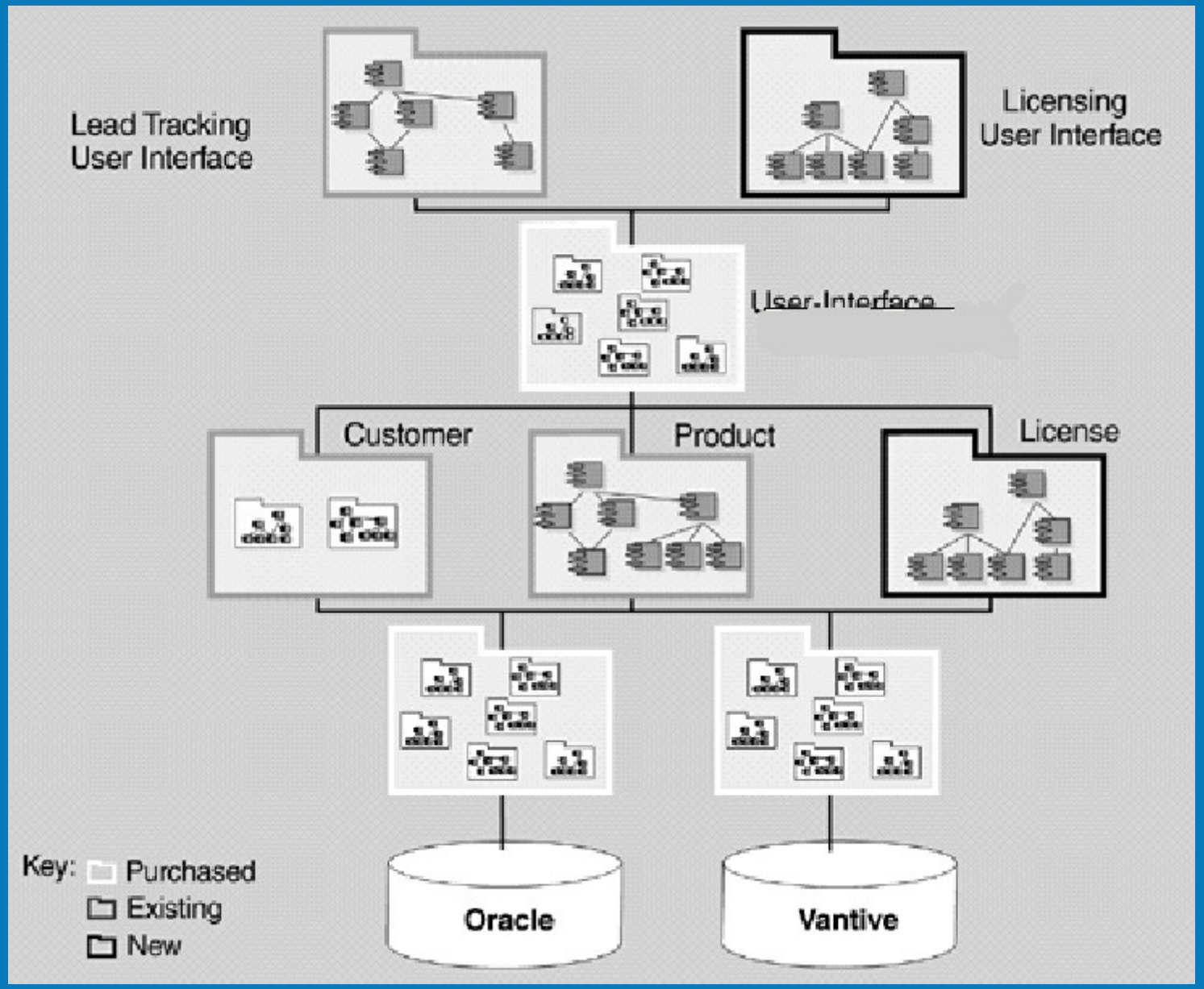


- Komponenty nigdy nie sprawują kontroli nad sobą – Zasada hollywoodzka: proszę do nas nie dzwonić, to my oddzwonimy...
- **Kontener** jest obiektem zarządzającym komponentami:
  - steruje procesem tworzenia komponentami
  - rozwiązuje zależności pomiędzy komponentami
  - zarządza cyklem życia komponentów

Cele architektury komponentowej

- **Podstawa dla ponownego użycia**
  - komponentów
  - architektury
- **Odpowiednie zarządzanie projektem poprzez:**
  - planowanie
  - Zatrudnienie odpowiedniej liczby osób w projekcie
  - Nadzorowanie dostaw poszczególnych komponentów
- **Kontrola**
  - zarządzanie złożonością
  - zapewnianie integralności

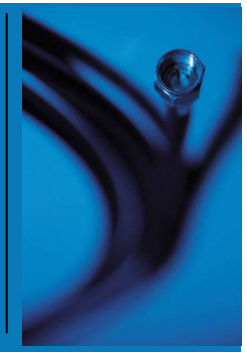



Lead Tracking
User Interface
Licensing
User Interface
User Interface
Customer
Product
License
Key:
Purchased
Existing
New
Oracle
Vantive

Praktyka 5: Stale weryfikuj jakość

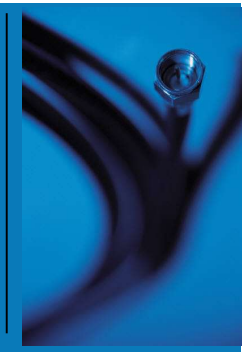
Jakość - właściwość opisująca wytworzony produkt, o którym, na podstawie uzgodnionych procedur i kryteriów pomiarowych, można powiedzieć, że zaspokaja (bądź przekracza) zdefiniowane dla niego wymagania, i został wytworzony w uzgodnionym procesie projektowym.

Definicja jakości



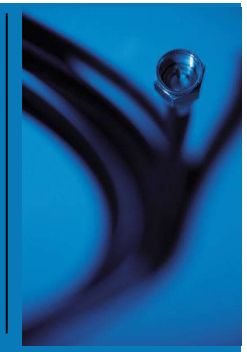
- Jakość produktu to nie tylko spełnianie przez niego wymagań.
- Jakość to również określenie procedur i kryteriów pomiaru jakości, oraz wdrożenie procesu, który zapewni, że produkt osiągnie zakładany poziom jakości (proces taki musi być również powtarzalny i zarządzany).

Zarządzanie jakością



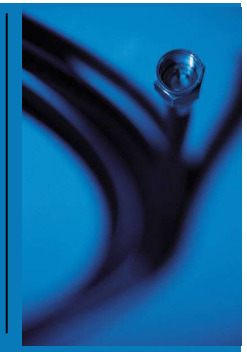
- W wielu przedsiębiorstwach testowanie oprogramowania pochłania od 30 do 50% całkowitych kosztów wytworzenia.
- Mimo tego, większość osób ma świadomość, że oprogramowanie nie jest odpowiednio przetestowane zanim zostanie wdrożone.

Zarządzanie jakością



- Podstawowe problemy:
  - testowanie oprogramowania jest skomplikowanym procesem. Liczba ścieżek przetwarzania, jakie może przejść działająca aplikacja, jest ogromna,
  - testowanie zazwyczaj odbywa się bez wyraźnie określonej metodologii oraz bez wystarczającego wsparcia ze strony automatycznych narzędzi. Często oprogramowanie jest tak złożone, że niemożliwe jest pełne go przetestowanie. W takim wypadku odpowiednia metodologia oraz automatyczne narzędzia mogą mocno zwiększyć efektywność procesu testowania.

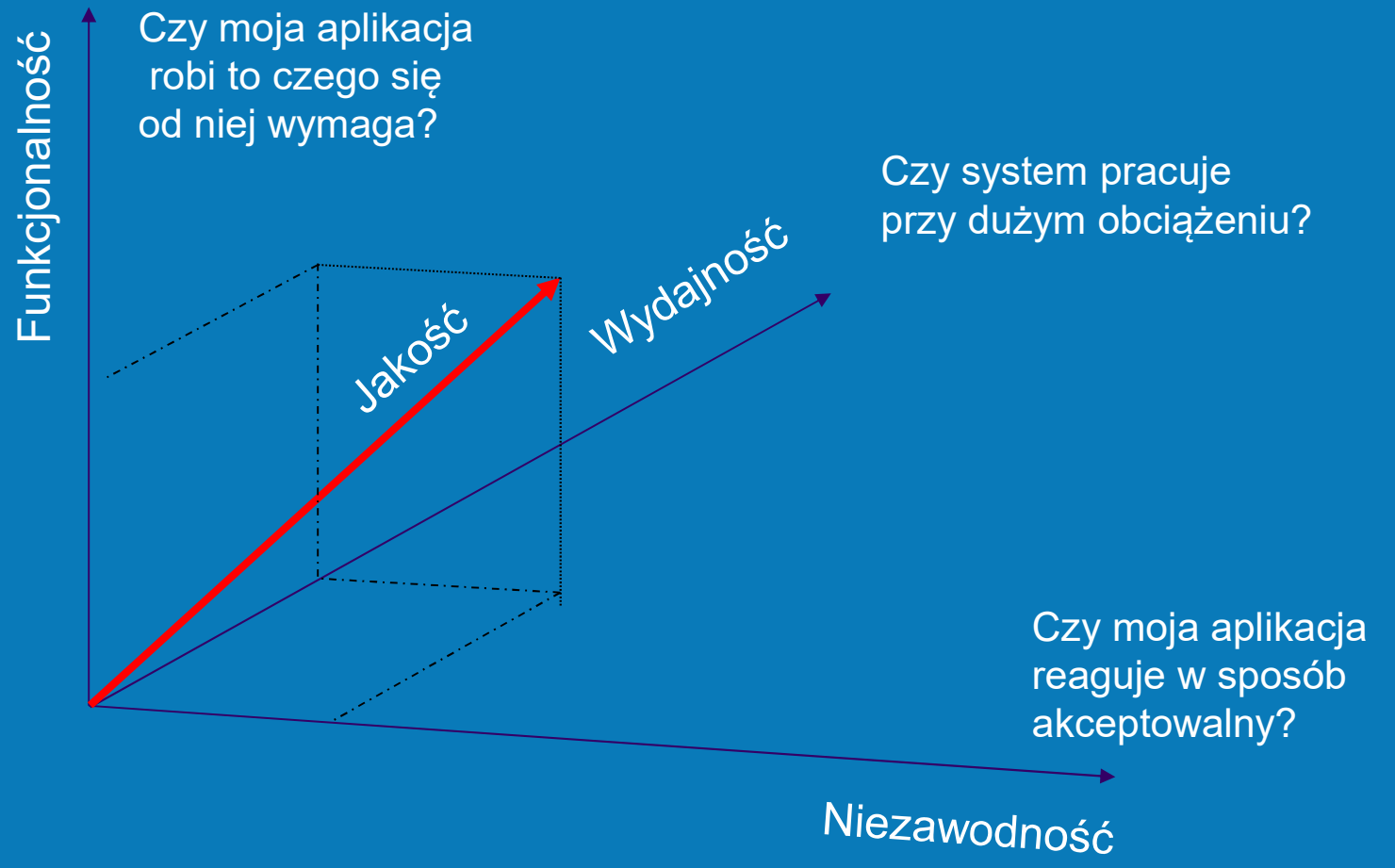
Ciągła weryfikacja jakości oprogramowania



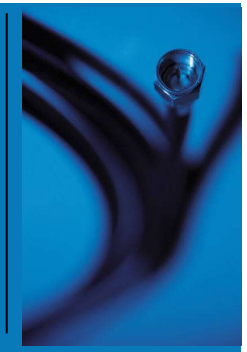
- Koszty naprawy błędów oprogramowania są od 100 do 1000 razy większe jeżeli wykryjemy je po wdrożeniu.
  - Koszty naprawy
  - Koszty straconych możliwości
  - Koszty utraty klientów

„Nie piszemy testów, bo nie mamy czasu.
Nie mamy czasu, bo poprawiamy bugi.”

Wymiary jakości oprogramowania



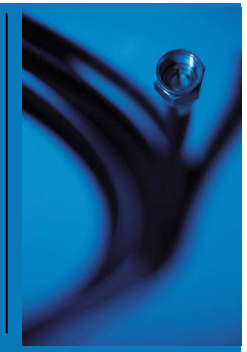
Weryfikacja wymiarów jakości



- Weryfikacja funkcjonalności – weryfikacja każdego scenariusza użycia. Wymaga stworzenia testu dla każdej instancji przypadku użycia (czyli dla każdego scenariusza).

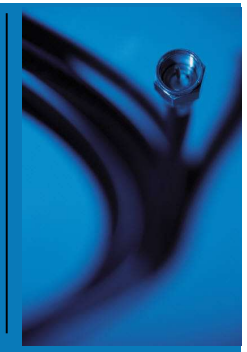
- Weryfikacja wydajności – testowanie przy obciążeniu nie tylko założonym, ale również najgorszym z możliwych.

Weryfikacja jakości



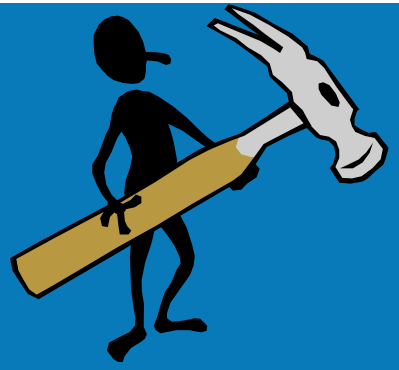
- Weryfikacja niezawodności – weryfikacja operacji wykonywanych w sposób ciągły:
  - Taka weryfikacja jest często wykonywana przez kompilatory i narzędzia do analizy kodu.
  - Najefektywniejsze techniki to: zautomatyzowane generowanie skryptów testowych oraz narzędziowo wspomagane wykonywanie testów regresyjnych (powtarzalne wykonywanie skryptów testowych).
  - Wykrywanie wycieków pamięci w języku C++ jest również efektywniejsze jeżeli korzystamy z odpowiednich narzędzi analizy programu.

Kilka uwag



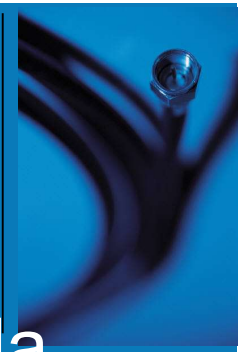
- **Słaba wydajność jest tak samo bolesna jak niska niezawodność.**
- Weryfikacja niezawodności – wąskie gardła, wycieki pamięci.
- Wszystkie typy weryfikacji mogą (powinny?) stanowić integralną część każdej iteracji w procesie wytwarzania.
- Testowanie przy każdej iteracji – najlepiej gdy jest wykonywane automatycznie.
- Testy dla podstawowych scenariuszy pomogą zapewnić, że wszystkie wymagania zostały odpowiednio zaimplementowane.

Testowanie w iteracjach



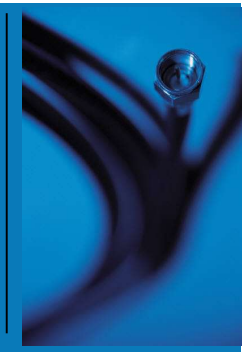
- W każdej iteracji testy powinny umożliwiać weryfikację czy spełnione są wymagania:
  - które były przedmiotem zainteresowania podczas iteracji,
  - które wynikają z wcześniejszych iteracji.
- Dzięki automatycznym narzędziom generowanie testów w kolejnej iteracji może być stosunkowo prostym zadaniem.

Praktyka 6: Zarządzaj zmianami



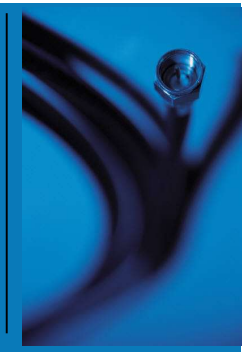
- W procesie wytwarzania zmiany zawsze będą się pojawiały.
  - **Równoczesne aktualizacje**
  - **Niepełne powiadamianie**
  - **Wiele wersji**
- Niezbędna jest kontrola tego gdzie zmiany zostają wprowadzane i kto je wprowadza.
- Niezbędna jest synchronizacja wprowadzanych zmian w całym zespole projektowym.

Zmiany w oprogramowaniu



- Dużym wyzwaniem procesu wytwarzania oprogramowania jest potrzeba współdziałania wielu developerów, podzielonych na zespoły, znajdujących się w różnych miejscach, pracujących razem nad wieloma iteracjami, wersjami, produktami, platformami.
- Jeżeli brakuje zdyscyplinowanego i kontrolowanego procesu taka sytuacja daje w wyniku tylko jedno – chaos.

Zmiany w oprogramowaniu

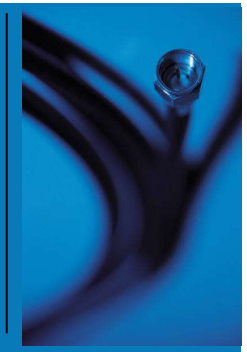


- Trzy najpowszechniejsze problemy to:
  - Równoczesne aktualizacje – kiedy dwóch lub większa liczba twórców oprogramowania jednocześnie aktualizuje ten sam artefakt procesu wytwarzania, to tylko ostatnia zmiana zostanie zapisana.
  - Niepełne powiadamianie – kiedy jakiś problem jest naprawiany – zmieniany jest współdzielony artefakt, niektóre zespoły nie zostają poinformowane o zmianie.
  - Wiele wersji – w procesie iteracyjnym pracowanie jednocześnie na więcej niż jednej wersji określonego artefaktu, z których każda znajduje się w innej fazie wytwarzania nie jest niczym niezwykłym. Należy zwrócić uwagę, że jeżeli w jednej z wersji zostaje wykryta usterka to poprawka musi być dokonana we wszystkich wersjach.

Co chcemy uzyskać?

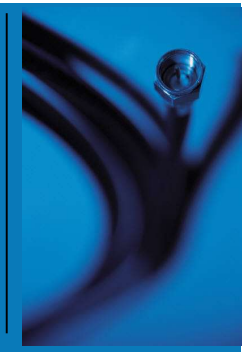
- Kontrolowanie zmian, aby umożliwić produkcję iteracyjną,
- Określanie przestrzeni roboczych dla każdego członka zespołu projektowego,
- Zarządzanie automatyczną integracją/generowaniem kodu wynikowego
- Równoległy proces wytwarzania.

Systemu zarządzania zmianami odpowiada za:



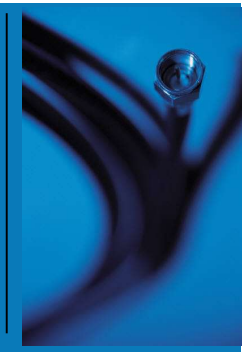
- Zarządzanie zgłoszeniami o zmianach w systemie
- Raportowanie statusu konfiguracji systemu
- Zarządzanie konfiguracją systemu
- Śledzenie zmian wykonywanych w systemie (powstawanie nowych skryptów)
- Wybór wersji
- Produkcja oprogramowania

Własności systemu zarządzania zmianami



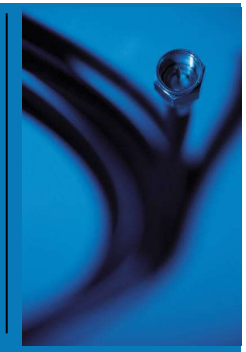
Zarządzanie zgłoszeniami – umożliwia szacowanie kosztów i harmonogramu wprowadzania zmiany. Szacowanie wpływu zgłoszonej potrzeby zmiany na istniejący produkt.

Własności systemu zarządzania zmianami



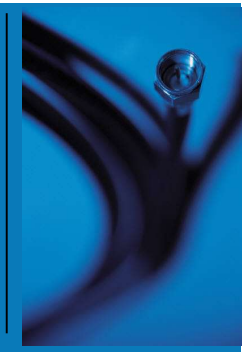
Raportowanie stanu konfiguracji – opis stanu produktu na podstawie typu, liczby oraz wagi usterek znalezionych i naprawionych w trakcie dotychczasowej produkcji. Oceny odbywają się bądź na zasadzie audytów lub na podstawie surowych danych. Metryki wypracowane w trakcie takich działań są bardzo użyteczne w procesie oceny kompletności produkcji.

Własności systemu zarządzania zmianami



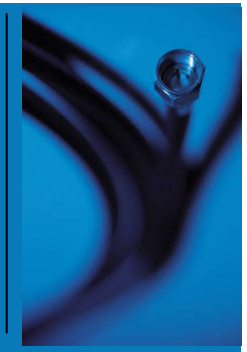
Zarządzanie konfiguracją – opis struktury produktu, jego elementów konfiguracyjnych, które są traktowane jako odrębne jednostki podlegające procesowi wersjonowania. Zarządzanie konfiguracją polega na: zdefiniowaniu konfiguracji, etykietowaniu oraz magazynowaniu artefaktów podlegających wersjonowaniu w taki sposób, aby możliwe było spójne zarządzanie związkami pomiędzy wersjami tego samego artefaktu.

Własności systemu zarządzania zmianami



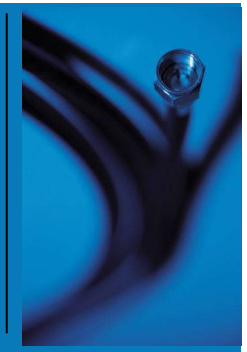
Śledzenie zmian –rejestrowanie historii wprowadzonych zmian. Jest to informacja całkowicie niezależna od informacji związanej z propozycją wprowadzenia nowej zmiany (patrz zarządzanie zgłoszeniami).

Własności systemu zarządzania zmianami



Wybór wersji – celem procesu wyboru wersji jest zapewnienie, że do procesu wprowadzania zmian została wybrana odpowiednia jednostka konfiguracyjna.

Własności systemu zarządzania zmianami

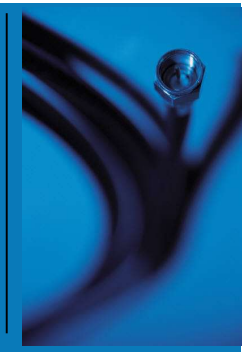


Produkcja oprogramowania – automatyzacja procesu kompilacji, testowania, tworzenia pakietów instalacyjnych oraz dystrybucji oprogramowania.

Wpływanie na siebie „Dobrych praktyk”

Wpływanie na siebie „dobrych praktyk”

Suma pojedynczych praktyk ma mniejszą wartość od wykorzystania wszystkich jednocześnie.

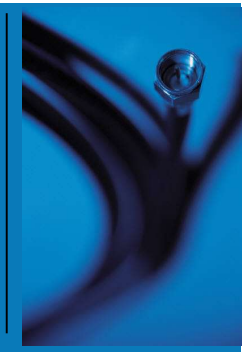


Spojrzenie na wytwarzanie systemów informatycznych z punktu widzenia procesów w nim zachodzących Rational Unified Process

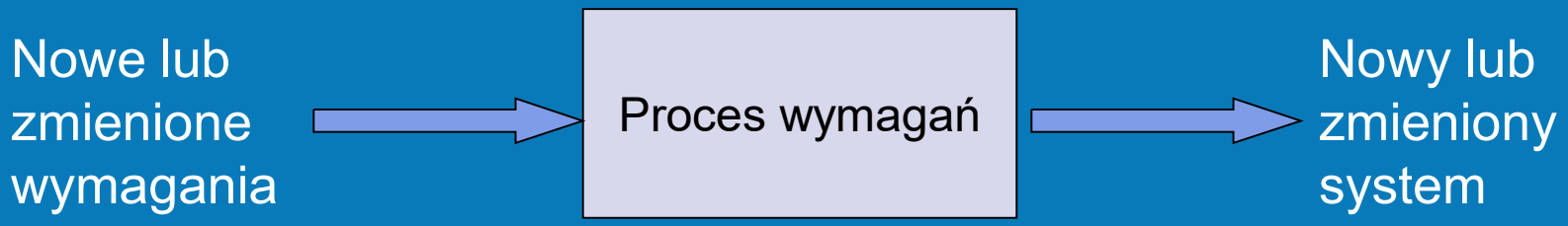
Podejście procesowe czyli: Po co potrzebna jest analiza procesów?

- Dostarcza wskazówek jak efektywnie wytworzyć oprogramowanie cechujące się wysoką jakością,
- Zmniejsza ryzyko popełniania błędów,
- Zwiększa przewidywalność projektu,
- Promuje wspólną wizję projektu
- Formalizuje dobre praktyki

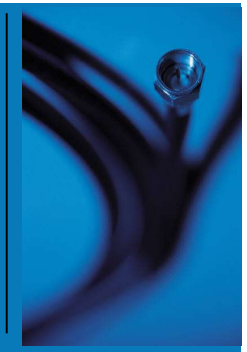
Definicja procesu w ujęciu zespołowym



Proces definiuje CO ma być zrobione, KIEDY, JAK i przez KOGO aby został osiągnięty określony cel.



Rational Unified Process (RUP)

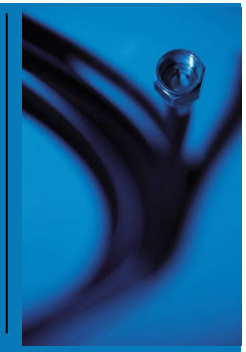


- Wykorzystuje podejście iteracyjne - zorientowane na architekturę, sterowane przypadkami użycia
- Jest to dobrze zdefiniowany proces inżynierii oprogramowania – kto, co, kiedy, jak
- Można go traktować jak przewodnik po działaniach i produktach (artefaktach)
- Wykorzystuje Przypadki Użycia sterujące projektem i implementacją
- Wykorzystuje modele, które tworzą abstrakcje systemu

Podejście RUP – podstawowe zasady

- Miej na względzie główne ryzyka projektu od początku procesu
- Uzyskaj pewność, że spełniasz wymagania użytkowników
- Miarą postępu jest działający kod
- Przystosuj się do zmian już we wczesnych fazach projektu
- Na wstępie zdefiniuj architekturę systemu
- Wykorzystuj architekturę komponentową
- Pracuj zespołowo
- Uczyń z jakości drogę a nie rezultat

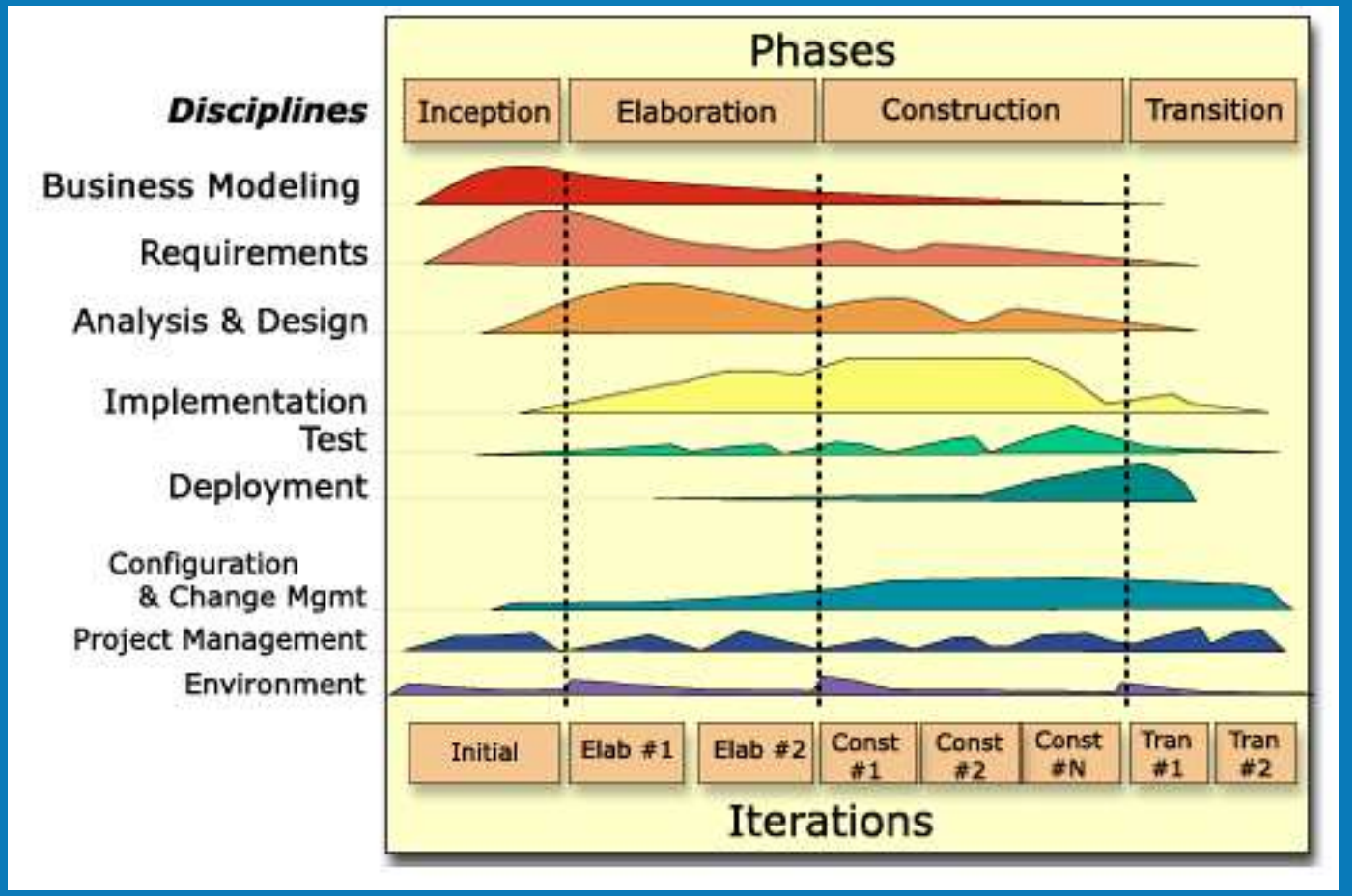
RUP – proces iteracyjny. Zalety:



- Dostosowany do zmian
- Integracja nie jest jednym wielkim procesem końcowym
- Ryzyka są wcześniej wykrywane
- Powtórne użycie
- Błędy mogą być wykryte w każdej iteracji
- Efektywniejsze wykorzystanie personelu

RUP – struktura dwuwymiarowa

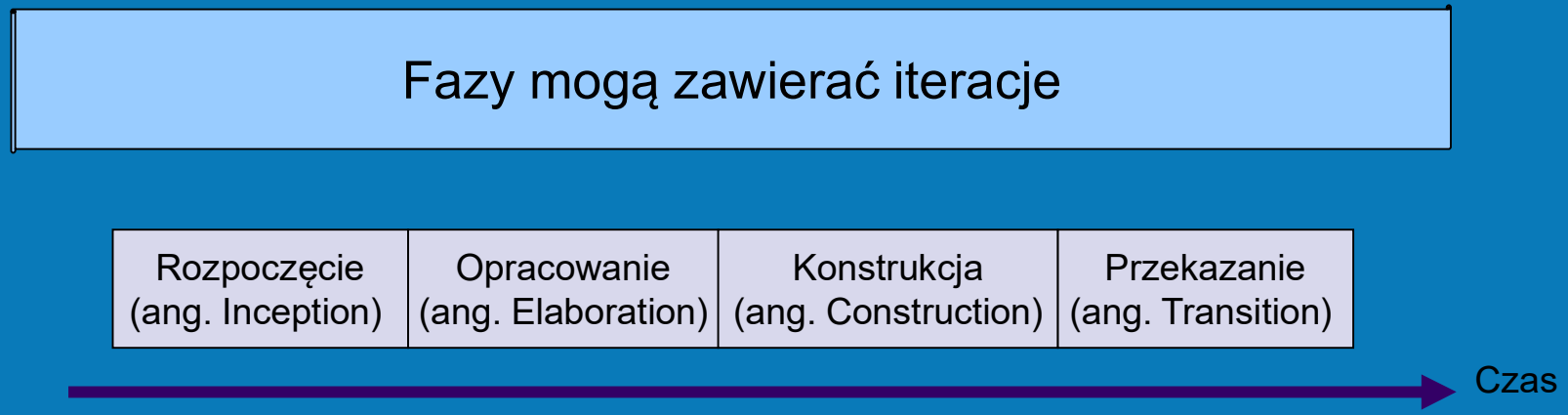
Wymiar struktury procesu: Jak elementy procesu (aktywności, artefakty, role, …) są logicznie pogrupowane w podstawowe dyscypliny procesu



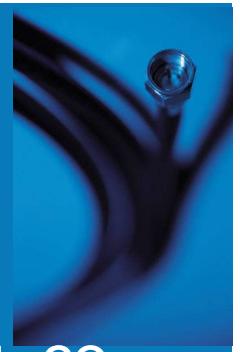
Wymiar czasu – cykle, fazy, iteracje, …

Dynamiczna struktura procesu RUP – fazy cyklu życia

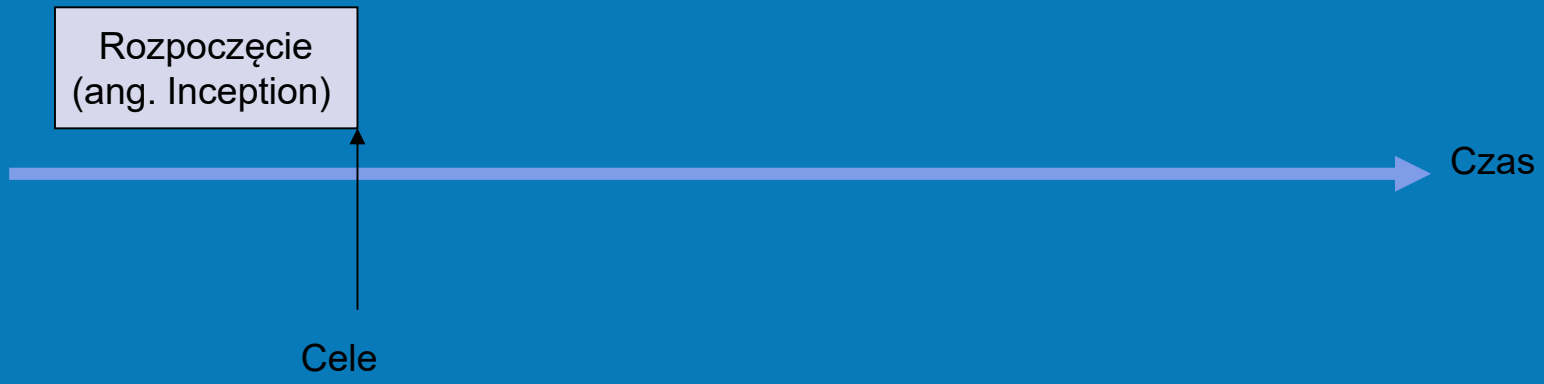
- **Rozpoczęcie** – Określenie założeń przedsięwzięcia
- **Opracowanie** – Ustalenie planu pracy i opracowanie dobrej architektury
- **Budowa** – Utworzenie systemu
- **Przekazanie** – udostępnienie systemu użytkownikom końcowym



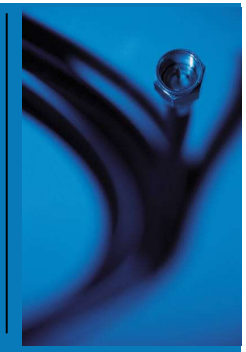
Rozpoczęcie (*ang. Inception*)



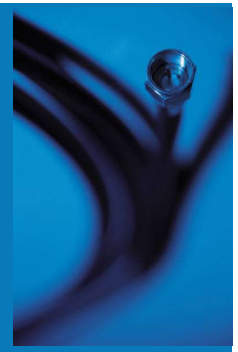
Podczas fazy **Rozpoczęcia** definiujemy zakres projektu, co zostanie włączone. Odbywa się to poprzez identyfikację aktorów i przypadków użycia oraz naszkicowanie najważniejszych modeli przypadków użycia (zwykle jest to ok.. 20% całości modelu). Równocześnie definiowany jest biznes plan w celu określenia czy i jakie zasoby należy przeznaczyć na poczet projektu.



Rozpoczęcie (*ang. Inception*)



- Cele:
  - Zrozumienie zakresu projektu (wizja produktu końcowego - czyli co jest celem a co nie)
  - Opracowanie „przypadków biznesowych” i ich prioryteryzacja
  - Przekonanie udziałowców do projektu
  - Określenie ogólnego harmonogramu projektu
  - Identyfikacja ryzyka i określenie sposobu ich minimalizacji
  - Organizacja środowiska produkcyjnego (wybór, instalacja i konfiguracja narzędzi).
- Kamień milowy:
  - Cele projektu (zbadanie celów przedsięwzięcia)

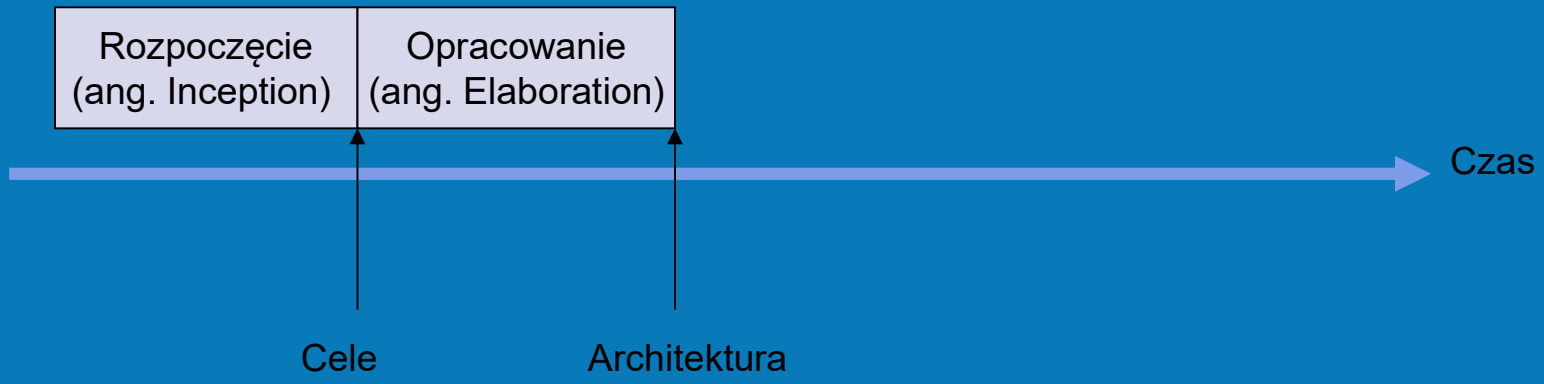


Opracowanie (*ang. Elaboration*)

W fazie **Opracowania** należy skupić się na dwóch rzeczach:

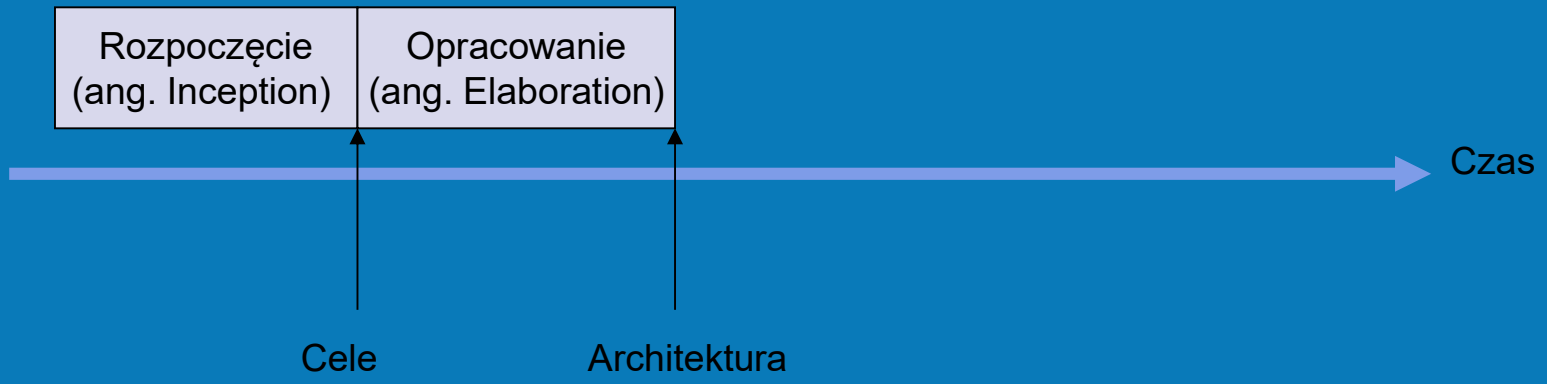
- zebraniu większości wymagań (c.a. 80%),
- dopracowaniu podstawy architektury.

Dzięki temu można wyeliminować znaczny procent ryzyka oraz określić z dużym prawdopodobieństwem ile pracy nam jeszcze pozostało. Pod koniec tej fazy jesteśmy w stanie dokonać bardziej szczegółowego oszacowania zasobów potrzebnych na dokończenie projektu.



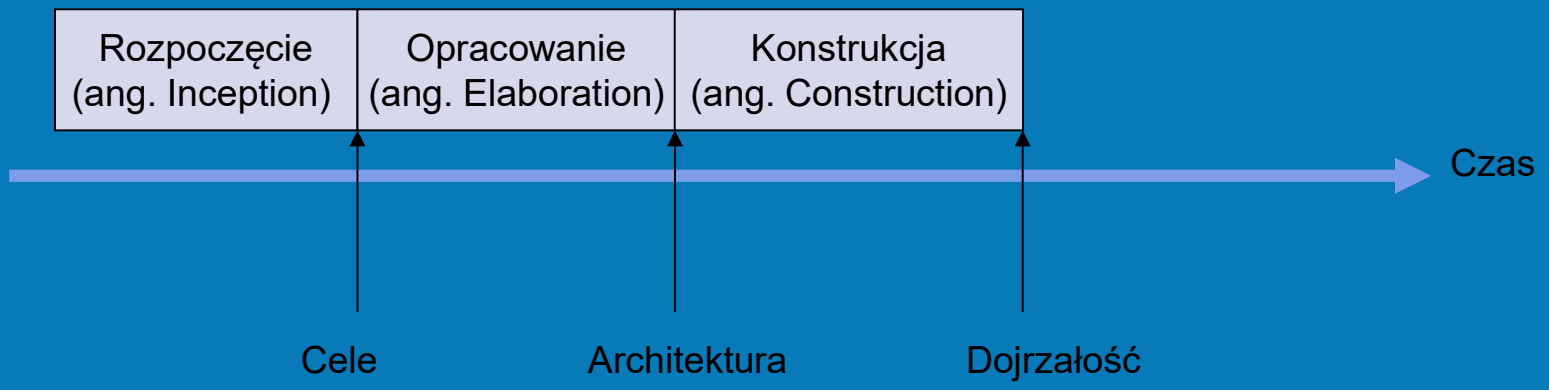
Opracowanie (*ang. Elaboration*)

- Cele:
  - Złagodzenie wpływu najistotniejszych ryzyk
  - Zdefiniowanie podstawowej architektury
  - Zrozumienie co jest potrzebne aby zbudować system
- Kamień milowy:
  - Architektura

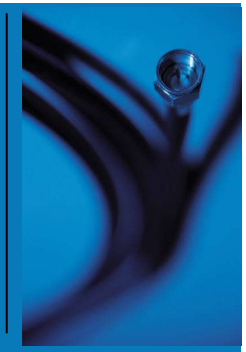


Budowa (*ang. construction*)

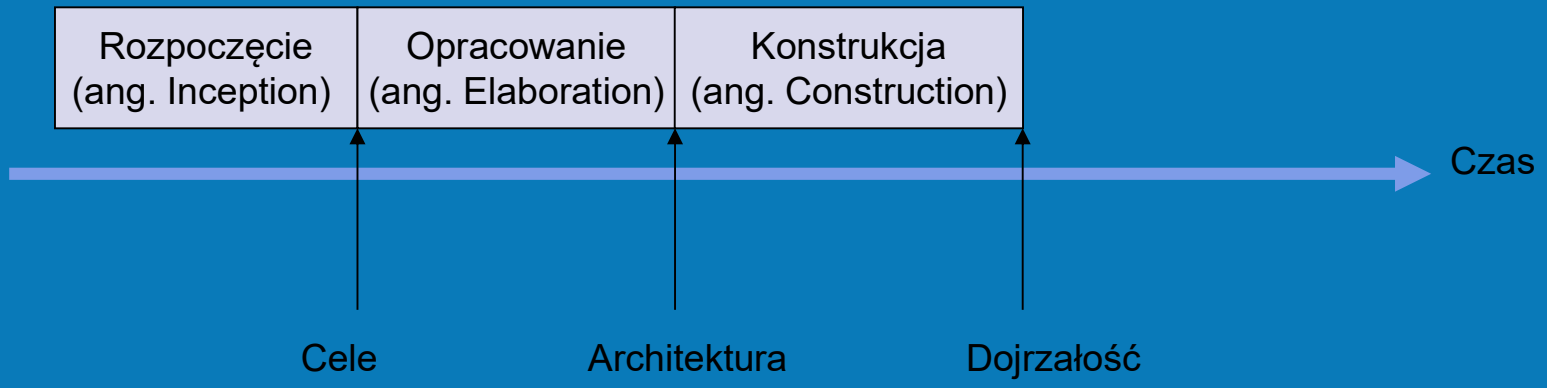
Podczas fazy **Budowy** konstruujemy system w kilku iteracjach aż do tzw. wydania wersji beta.



Budowa (*ang. construction*)

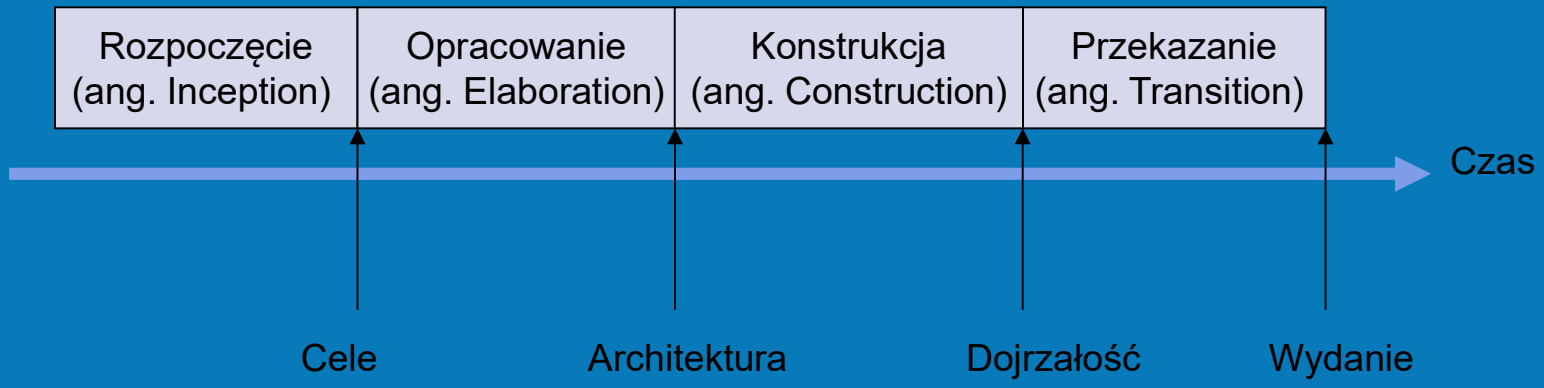


- Cele:
  - Zbudowanie pierwszej działającej wersji systemu
    - Modyfikacja wizji, architektury, planów
- Kamień milowy:
  - Dojrzałość operacyjna (zapewnienie sprawnego działania początkowej wersji)



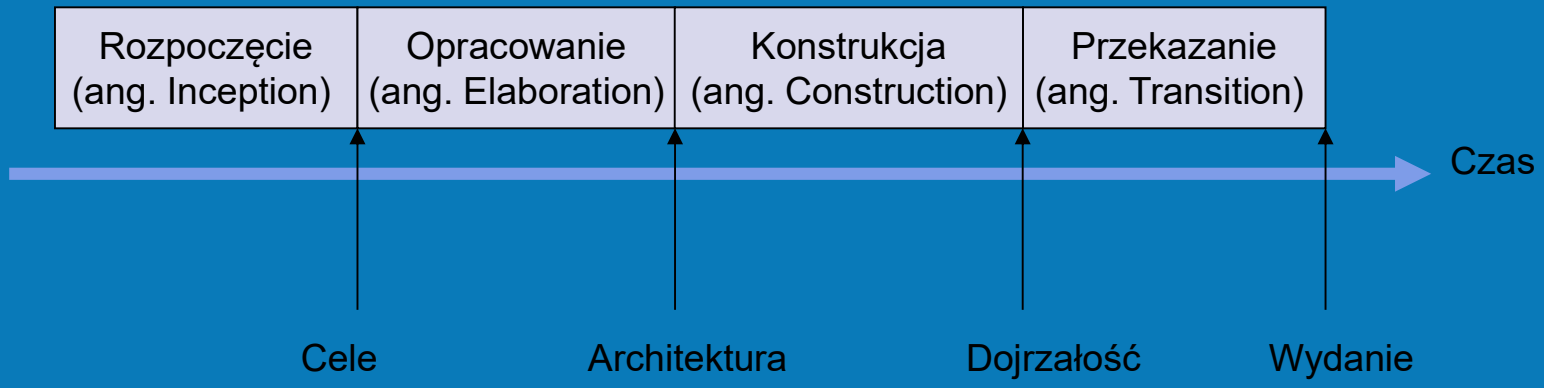
Przekazanie (*ang. transition*)

W fazie **Przekazania** produkt jest przekazywany użytkownikom. Nacisk położony jest na szkolenia użytkowników, instalację oraz wsparcie.



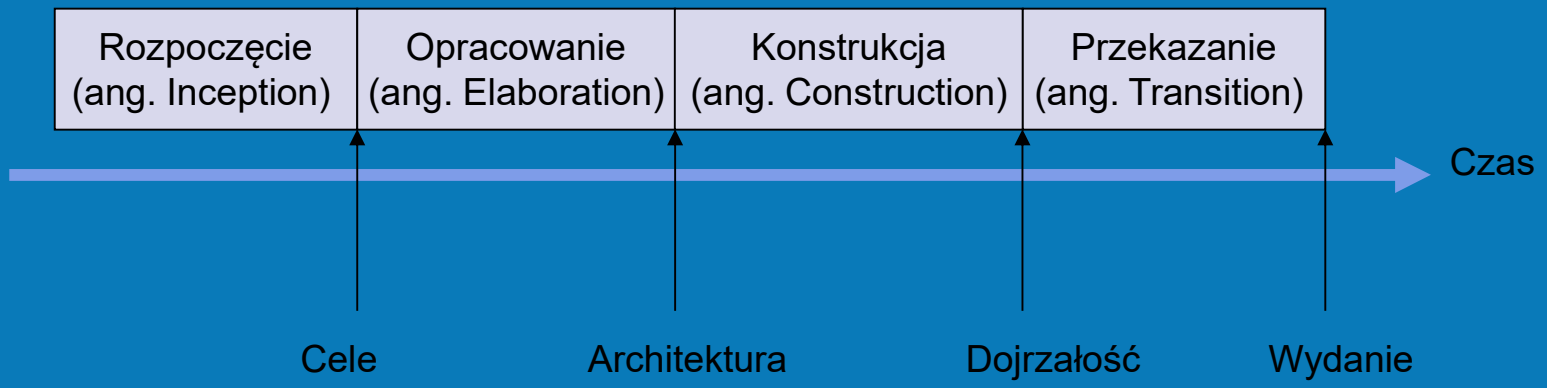
Przekazanie (*ang. transition*)

- Cele:
  - Zbudowanie wersji finalnej i przekazanie jej klientowi
  - Szkolenie użytkowników, nadzór autorski, opieka
  - Tuningowanie kodu i poprawa błędów
- Kamień milowy:
  - Wydanie (ang. release) – udostępnienie produktu klientowi



Granice faz – kamienie milowe

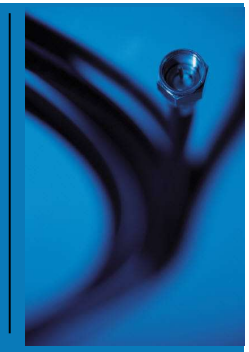
- Granice faz definiują główne kamienie milowe projektu

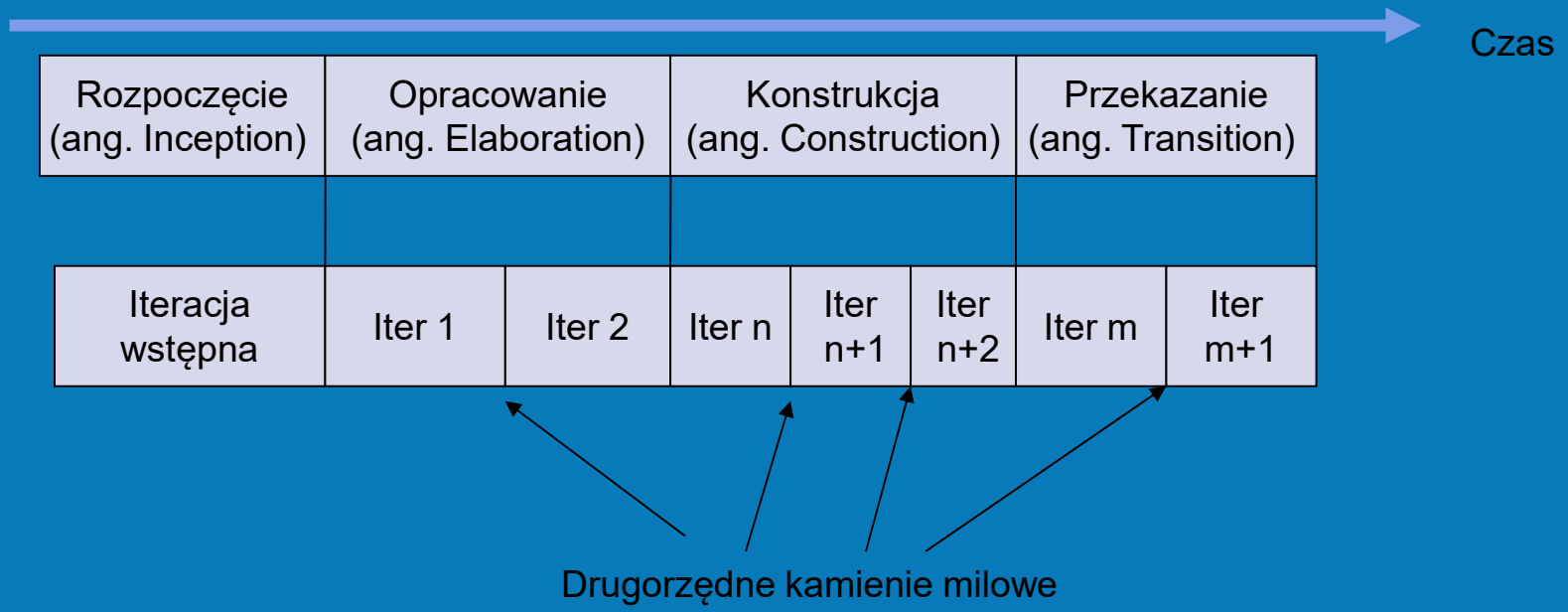


Granice faz – kamienie milowe

- Każdy z głównych kamieni milowych jest punktem czasowym, w którym dokonujemy oceny pewnych aspektów związanych z procesem projektowym.
- Od wyniku oceny zależy czy projekt dalej będzie kontynuowany, czy też należy poddać go korekcie lub porzucić. Kryteria oceny są różne dla różnych faz.
- Czas jaki powinno poświęcić się każdej z faz zależy od samego projektu. Dla bardzo złożonych projektów faza Opracowania może mieć nawet do 5 iteracji, natomiast w projektach, dla których wymagania są dobrze znane oraz architektura jest prosta może ona mieć tylko jedną iterację.

Iteracje i fazy





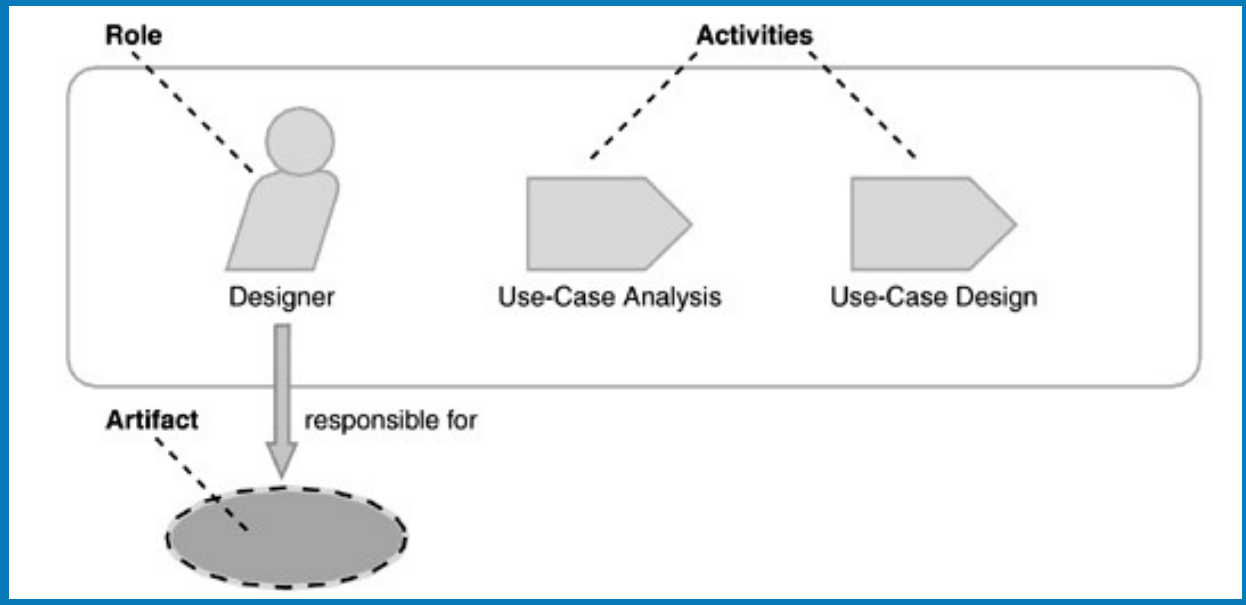
- Iteracja jest odrębną sekwencją działań opierającą się na zatwierdzonym planie oraz określonych kryteriach oceny, której wynikiem jest działające wydanie (zewnętrzne bądź wewnętrzne)

Iteracje i fazy

- Każda z faz składa się z serii iteracji. Liczba iteracji w określonej fazie jest zmienna. Każda z iteracji daje w wyniku działające wydanie, które zawiera w sobie wzrastający podzbiór ostatecznej funkcjonalności aplikacji.
- **Wewnętrzne wydanie** działa tylko w środowisku wytwórczym i może być zademonstrowane klientom. Użytkownicy otrzymują dopiero **wydanie zewnętrzne**, które jest instalowane w środowisku docelowym. Wydania zewnętrzne pojawiają się dopiero podczas fazy przekazania.
- Każda iteracja kończy się oceną rezultatów i ewentualną korektą przyszłych planów.

Struktura statyczna RUP

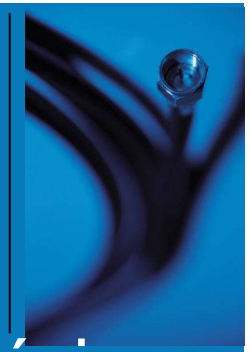
Jak elementy procesu (aktywności, artefakty, role, …) są logicznie pogrupowane w podstawowe dyscypliny procesu



Proces definiuje **co** ma być zrobione, **przez kogo**, **jak** i **kiedy**

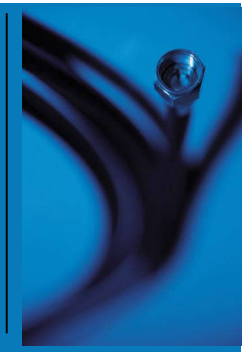
Cztery podstawowe elementy modelu RUP:
Role – definiują **kto** jest odpowiedzialny
Aktywności (czynności) – **jak**
Artefakty (i.e. elementy projektowe) – **co**
Przepływy prac (ang. workflow) - **kiedy**

Role



- Określa zachowanie i zakres odpowiedzialności
  - Osoby lub grupy osób (zespołu)
- Przykłady ról:
  - Analityk systemu:
    - prowadzi i koordynuje zbieranie wymagań i modelowanie przypadków użycia
  - Projektant:
    - Wyznacza zakres odpowiedzialności, atrybuty, operacje, związki dla klas, dostosowuje je do implementacji
  - Projektant testów
    - Planowanie, projektowanie, implementację i ocenę testów

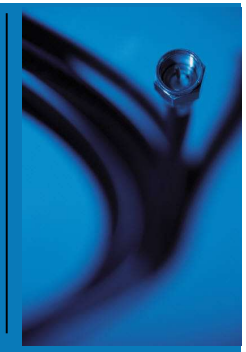
Aktywność/czynność



- Ważne jest określenie jednostki czasu pracy dla określonej roli
  Cel: Powinna nadawać się do użycia w planowaniu (miernik postępu)

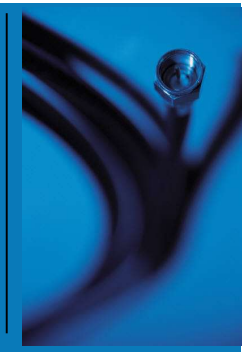
- Przykłady:
  - Planowanie iteracji : rola – kierownik przedsięwzięcia
  - Wyznaczenie przypadków użycia i aktorów : rola – analityk systemu

Czynności



- Zrozumienie istoty zadania
  - Zebranie i przeanalizowanie artefaktów wejściowych
- Wykonywanie
  - Tworzenie lub aktualizacja określonych artefaktów
- Recenzowanie
  - Badanie wyników na podstawie określonych kryteriów

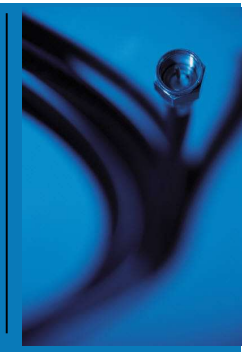
Artefakty



- Porcja informacji tworzona, modyfikowana lub używana przez proces
- Konkretne produkty przedsięwzięcia
- Wejścia i wyjścia dla czynności procesu

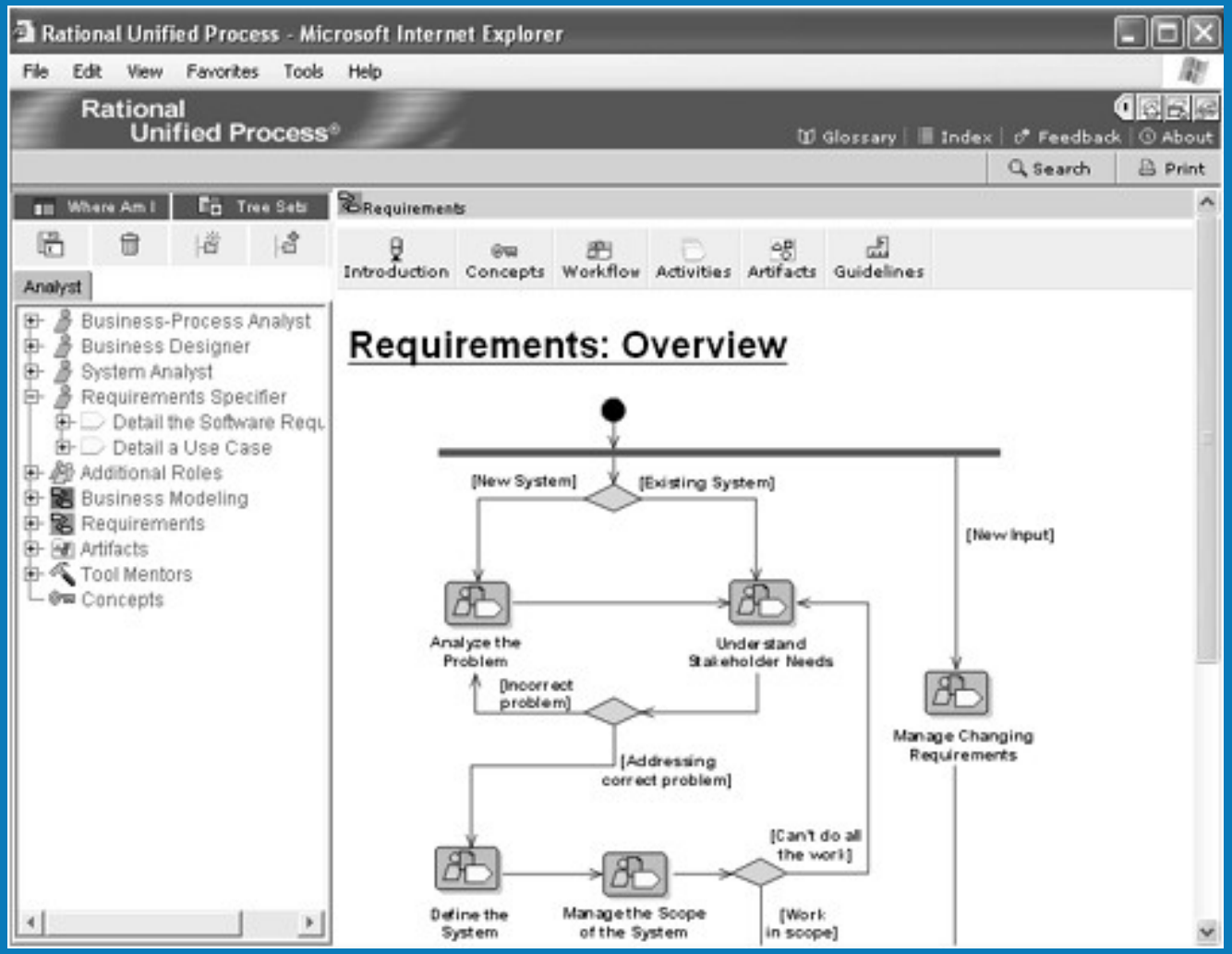
Mogą podlegać kontroli wersji i zarządzaniu konfiguracją

Przepływy prac (ang. Workflow)



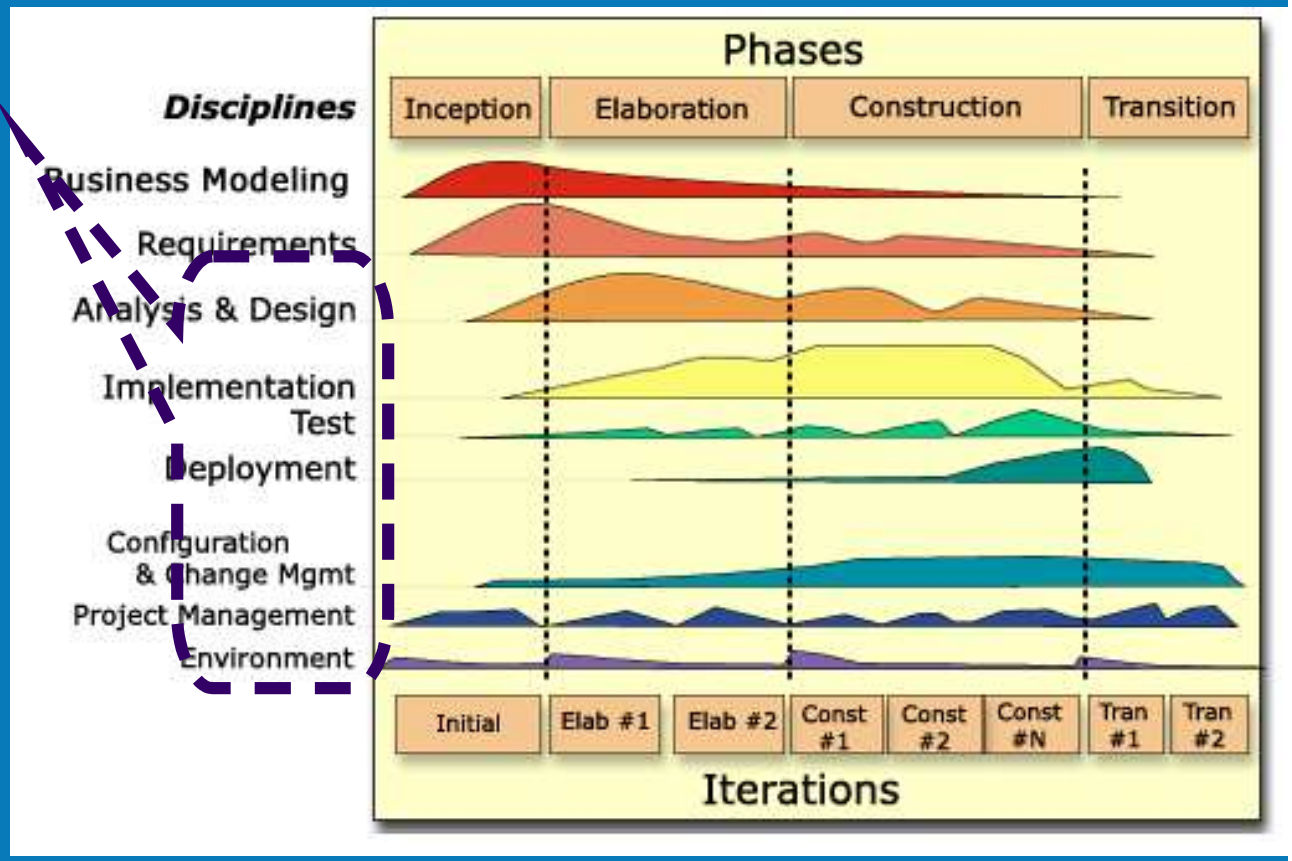
- Sekwencja czynności, których wynik ma zauważalną wartość
- RUP definiuje trzy typy przepływów prac:
  - Podstawowe – przyporządkowane każdej dyscyplinie
  - Szczegółowe – drobiazgowe przedstawienie przepływów podstawowych
  - Planowanie iteracji

Przepływy prac



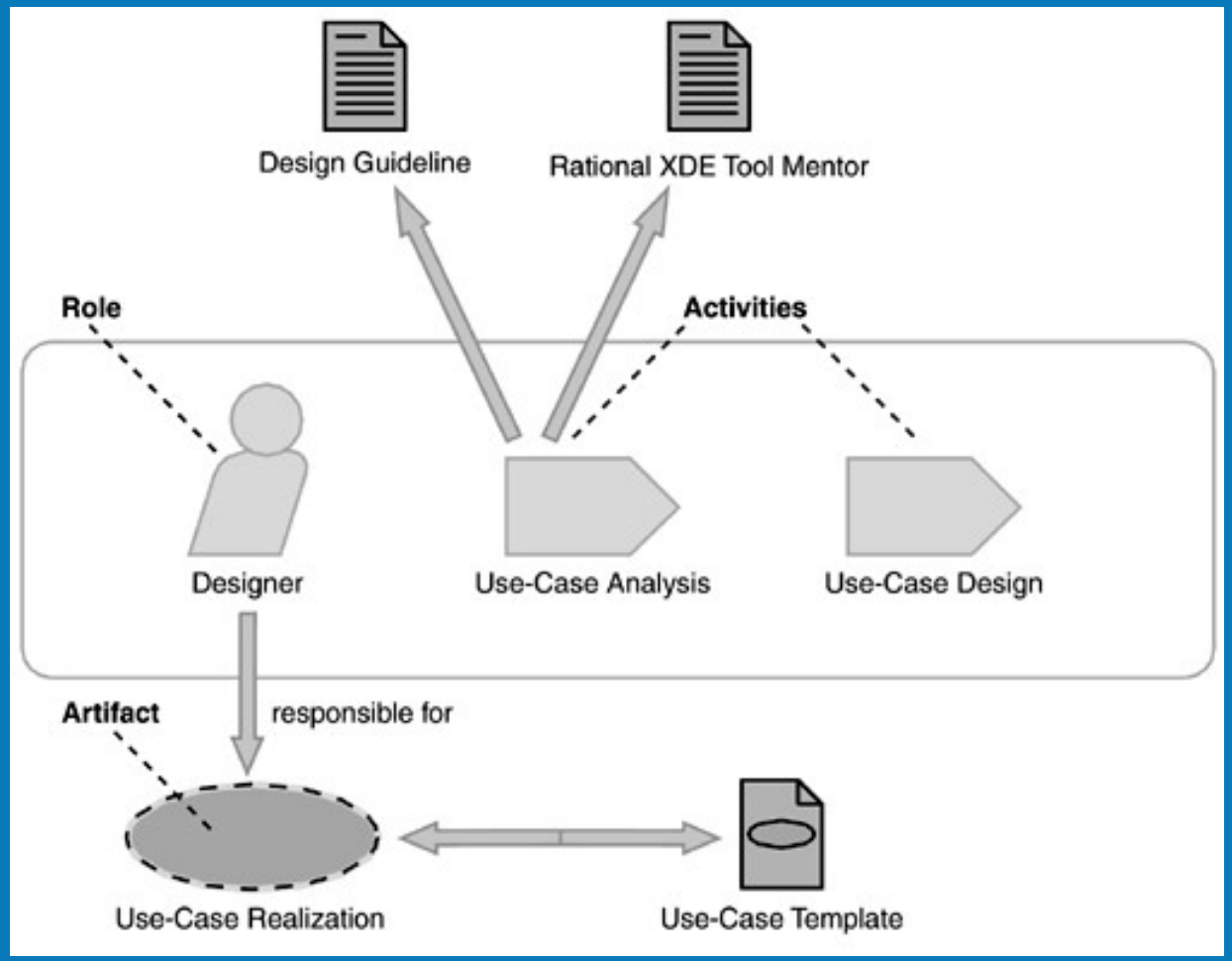
Dyscypliny

Elementy procesu są pogrupowane w dyscypliny

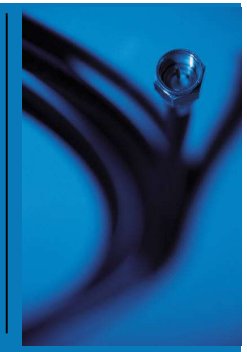


Inne elementy procesu

Wskazówki
Szablony
Definicje
Wsparcie narzędziowe
Przewodniki



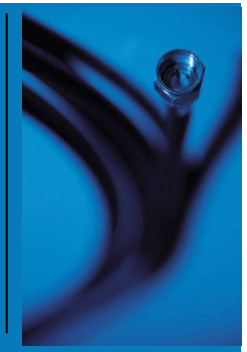
Wskazówki

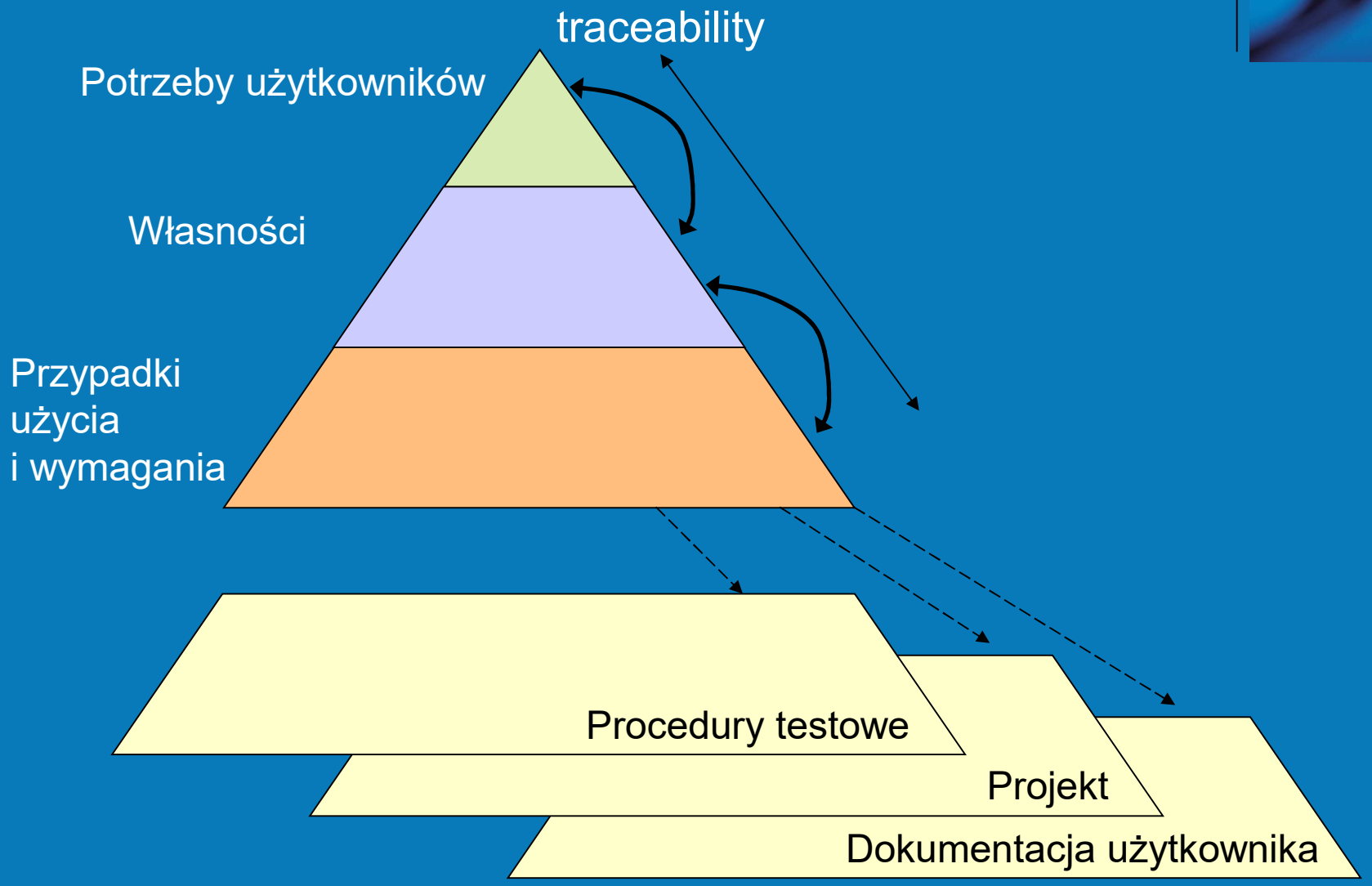


Reguły, zalecenia, heurystyki dołączane do czynności, kroków, artefaktów opisują:

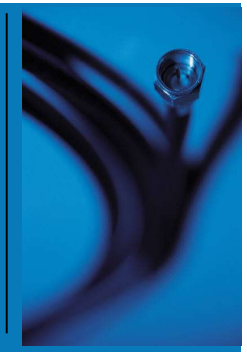
- poprawnie zbudowane artefakty
- konkretne techniki tworzenia artefaktów lub ich transformacji
- Sposoby oceny jakości artefaktów (listy kontrolne)

Zarządzanie wymaganiami i „traceability”





Podsumowanie



- Dobre praktyki – doświadczenie płynące z projektów
- Rational Unified Process (RUP)
  - Iteracyjne, zorientowane na architekturę, sterowane **przypadkami użycia** podejście do wytwarzania oprogramowania
  - Dobrze zdefiniowany proces inżynierii oprogramowania