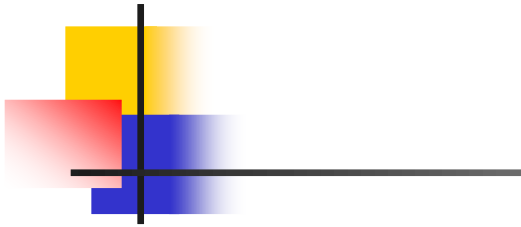
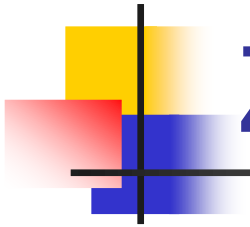
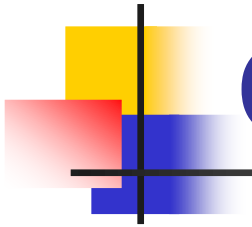


Pojęcie niezawodności systemów informatycznych



Zagadnienia

- Definicja niezawodności
- Niezawodność a wydajność
- Metryki niezawodności
- Specyfikacje niezawodności
- Testowanie statystyczne, wzorce operacyjne
- Techniki uzyskiwania niezawodności



Czym jest niezawodność?

Czym jest niezawodność?

- Prawdopodobieństwo z jakim system będzie bezawaryjnie funkcjonował w zadanym okresie czasu, przy pracy w określonym środowisku i dla określonego celu
  - Oprogramowanie systemu lotu – 99.99% w czasie 1 lotu
- Ma to całkowicie różne znaczenia w zależności od systemu oraz jego sposobu wykorzystywania
- Mniej formalnie niezawodność to miara tego w jakim stopniu w odczuciu użytkowników system dostarcza oczekiwanych od niego usług

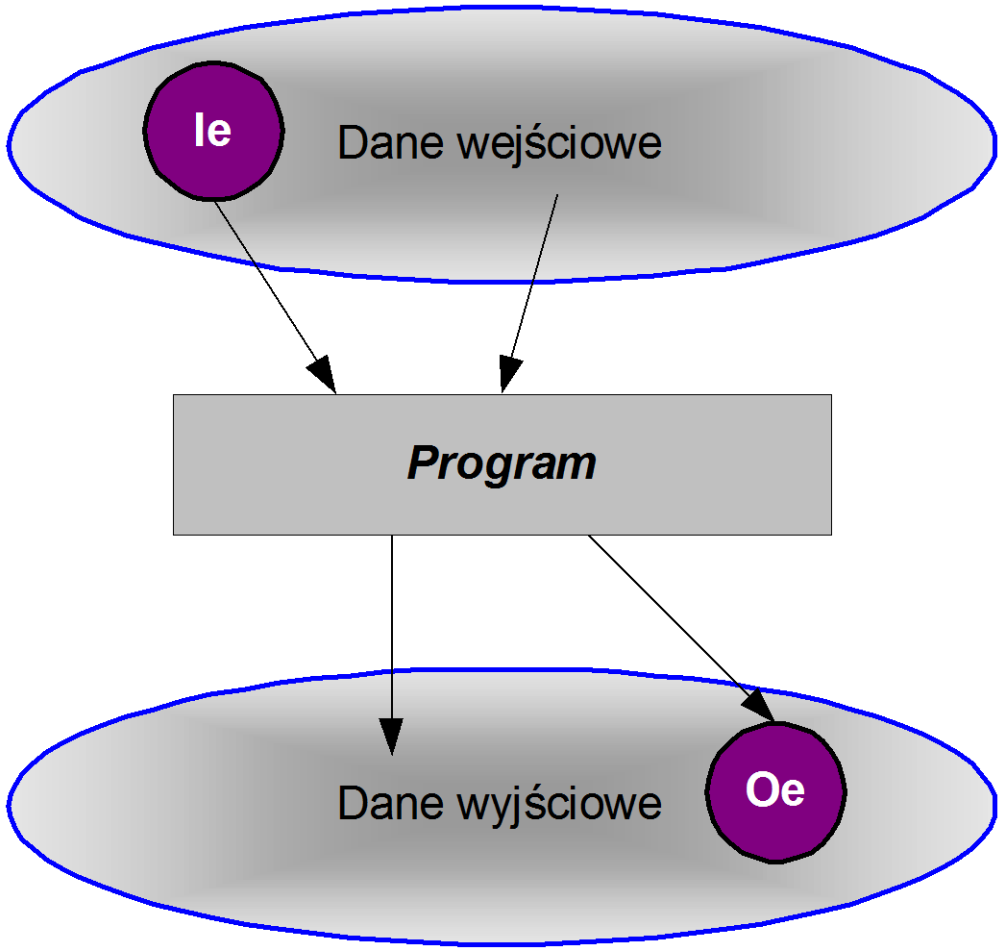
Cechy niezawodności

- Nie da się zdefiniować generalnie
  - Miary niezawodności sformułowane bez odniesienia do kontekstu są bezwartościowe
- W celu zdefiniowania – trzeba określić tzw. wzorzec operacyjny
  - Wzorzec taki określa sposób w jaki oprogramowanie będzie użytkowane
- Rozpatruje się ją biorąc pod uwagę konsekwencje awarii
  - Awarie mają rozmaite konsekwencje. Im mniej występuje poważnych awarii, tym bardziej dany system jest postrzegany jako niezawodny.

Awarie a błędy

- Awaria to nieoczekiwane zachowanie się systemu które wystąpiło w czasie jego pracy i zostało zauważone przez użytkowników
- Błąd jest cechą statyczną oprogramowania – awarie są konsekwencjami błędów
  - Błędy programistyczne, projektowe
  - Wykrywane przez inspekcje a także - obserwacje awarii
- Błędy niekoniecznie powodują awarie systemu. Dzieje się tak jeśli zostanie użyta część oprogramowania zawierająca błąd

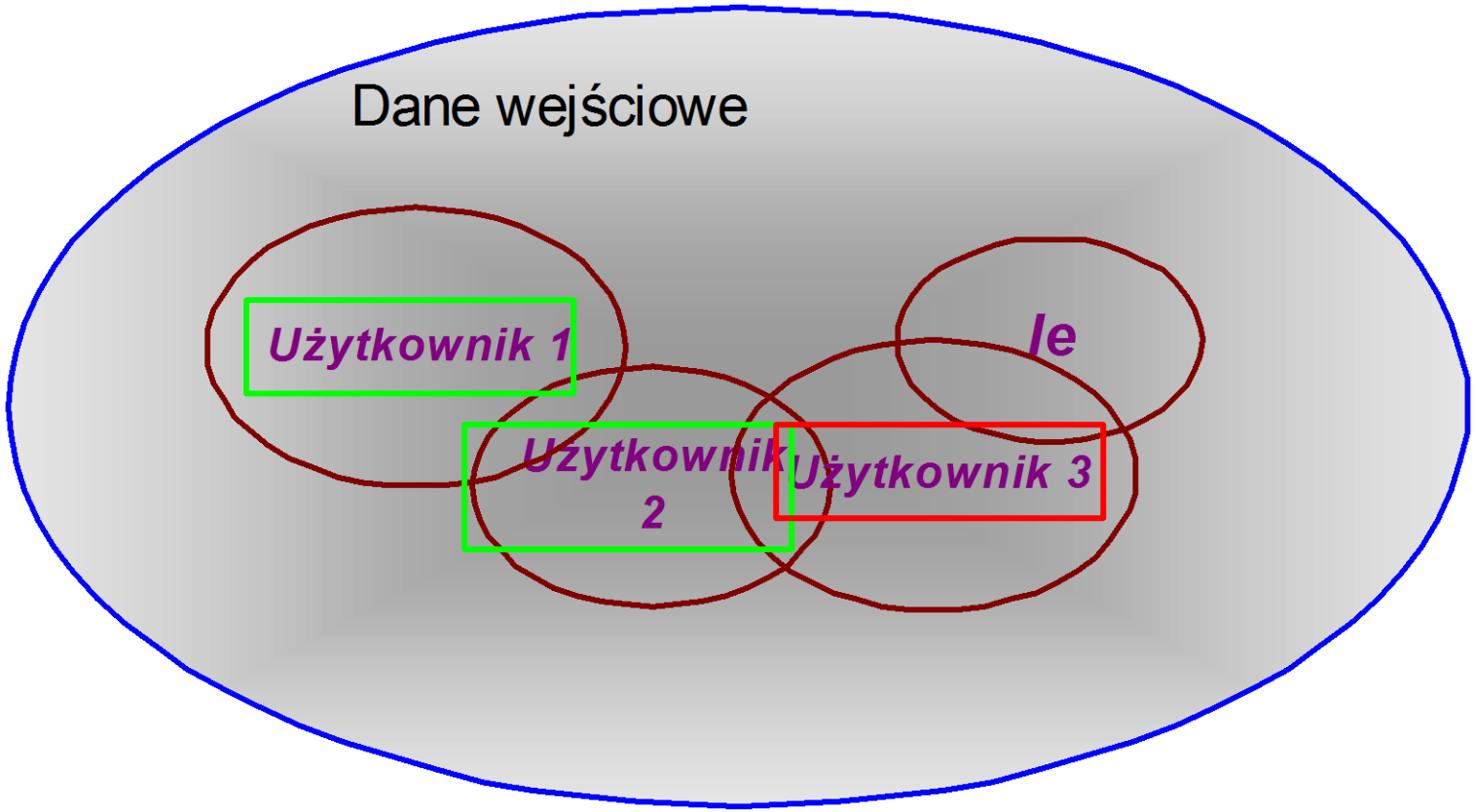
Mapowanie We ⇒ Wy



Poprawianie niezawodności

- Ma miejsce wówczas gdy usuwa się błędy występujące w najczęściej używanych fragmentach oprogramowania
- Usunięcie np. 10% błędów w oprogramowaniu wcale nie oznacza 10% poprawy niezawodności
- Wg Mills et. al. usunięcie 60% defektów spowodowało zaledwie 3% wzrost niezawodności
- Najważniejszym celem jest wyeliminowanie błędów mogących mieć poważne konsekwencje
- System może być uważany przez użytkowników za niezawodny mimo tego że zawiera błędy
  - Objawienie się -> tylko w specyficznych warunkach

Postrzeganie niezawodności przez użytkowników



Niezawodność a wydajność (1/2)

- Zazwyczaj wzrost niezawodności systemu prowadzi do odniżenia jego wydajności
- W celu zwiększenia niezawodności systemu dodaje się nadmiarowy kod do dynamicznej weryfikacji różnych warunków w czasie wykonywania programu. Powoduje to wzrost czasu wykonania

Niezawodność a wydajność (2/2)

- Zwykle niezawodność jest ważniejszą cechą systemu niż jego wydajność
- Zawodne oprogramowanie
  - Nie jest wykorzystywane – obawa przed konsekwencjami
  - Jest trudne w pielęgnacji/rozwijaniu
- Koszt awarii systemu niejednokrotnie przekracza koszt jego wytworzenia
- Koszty utraty danych są szczególnie wysokie

Miary niezawodności

- Miary stosowane w odniesieniu do hardware'u są nieprzydatne dla oprogramowania
  - Oparte o niezawodność komponentów które w razie awarii można zastąpić nowymi
  - Zakłada się tu że projekt jest poprawny
- Awarie oprogramowania są zawsze konsekwencjami błędów w projekcie
- Często system kontynuuje działanie mimo wystąpienia awarii – w przeciwieństwie do systemów hardware'owych

Miary niezawodności

- Prawdopodobieństwo awarii przy żądaniu usługi (*ang. probability of failure on demand – POFOD*)
  - Miara prawdopodobieństwa wystąpienia awarii przy zażądaniu usługi od systemu
  - POFOD = 0.005 oznacza że na każde 1000 żądań 5 z nich zakończy się awarią
  - Stosowana dla systemów o działaniu ciągłym i wysokiej niezawodności
- Odsetek wystąpień awarii (*ang. rate of fault occurrence - ROCOF*)
  - Częstość wystąpienia nieoczekiwanego zachowania systemu
  - ROCOF 0.01 oznacza że w czasie każdych 100 jednostek czasowych funkcjonowania systemu zdarzy się 1 awaria
  - Stosowana dla systemów operacyjnych oraz dla systemów przetwarzania transakcyjnego

Miary niezawodności

- Średni czas wystąpienia awarii (*ang. mean time to failure – MTTF*)
  - Czas pomiędzy zaobserwowanymi awariami
  - MTTF = 200 oznacza że średni czas pomiędzy zaobserwowanymi awariami wynosi 200 jednostek czasowych
  - Stosowana dla systemów z długimi transakcjami
- Dostępność (*ang. availability*)
  - Miara czasu ciągłej dostępności systemu – uwzględnia czas potrzebny na restart bądź naprawę systemu po awarii
  - Dostępność = 0.998 oznacza że system jest dostępny dla użytkowników w 998 z każdych 1000 jednostek czasowych
  - Stosowana dla systemów pracy ciągłej, np. central telefonicznych

Ocena niezawodności systemu

- Pomiar liczby awarii systemu dla zadanej liczby danych wejściowych
  - Służy do obliczenia POFOD
- Pomiar czasu (bądź liczby transkacji) pomiędzy awariami systemu
  - Służy do obliczenia ROCOF oraz MTTF
- Pomiar czasu potrzebnego na ponowne uruchomienie systemu po wystąpieniu awarii
  - Służy do obliczenia dostępności (ang. AVAIL)

Jednostki czasu

- Jednostki czasu używane przy obliczaniu miar niezawodności powinny być dostosowane do rodzaju danego systemu
  - Czas pracy procesora (dla systemów pracy ciągłej – np. systemy central telefonicznych – większość czasu spędzają na oczekiwaniu na zgłoszenie)
  - Czas kalendarzowy (dla systemów o stałym wzorcu operacyjnym, takich jak np. system księgujący transakcje uruchamiany raz dziennie o stałej porze; systemy alarmowe)
  - Liczba transakcji (dla systemów o chwilowym dużym obciążeniu wieloma transakcjami – np. bankomaty czy systemy rezerwacji biletów lotniczych)

Konsekwencje awarii

- Miary niezawodności nie uwzględniają konsekwencji ewentualnych awarii
- Przejściowe (*ang. transient*) błędy na ogół nie powodują żadnych poważnych komplikacji; inne typy błędów mogą spowodować utratę ważnych danych bądź usług oferowanych przez system
- Przy tworzeniu specyfikacji niezawodności systemu należy dla poszczególnych kategorii awarii definiować odrębne miary

Specyfikacja niezawodności

- Wymagania dot. niezawodności w praktyce są bardzo rzadko wyrażane w weryfikowalny sposób
- Aby możliwa była weryfikacja danej metryki niezawodności, należy jako część planu testów podać wzorzec operacyjny
- Niezawodność jest cechą dynamiczną systemu – częsty błąd popełniany przy specyfikowaniu wymagań dot. niezawodności to definiowanie metryk w oparciu o kod źródłowy
  - Max. N błędów na 1000 linii kodu
  - Skąd wiadomo czy wykryto wszystkie błędy?

Klasyfikacja awarii

Klasa	Opis
Przejściowe	Pojawiają się tylko dla niektórych danych wejściowych
Trwałe	Pojawiają się dla wszystkich danych wejściowych
Odwracalne	System jest w stanie kontynuować działanie bez interwencji operatora
Nieodwracalne	Do kontynuacji działania systemu jest konieczna interwencja operatora
Nieniszczące	Wystąpenie awarii nie niszczy danych ani stanu systemu
Niszczące	Wystąpienie awarii powoduje zniszczenie danych bądź stanu systemu

Klasy awarii można również definiować jako kombinacje powyższych cech.

Tworzenie specyfikacji niezawodności

- Dla każdego podsystemu przeanalizuj konsekwencje wystąpień możliwych awarii
- Podziel awarie powstałe w wyniku analizy na odpowiednie klasy
- Dla każdej zidentyfikowanej klasy wyspecyfikuj wymagania dotyczące niezawodności używając odpowiednich miar. Dla różnych wymagań mogą zostać wykorzystane różne metryki

Przykład: sieć bankomatów

- Każdy bankomat w sieci jest używany ~ 300 razy dziennie
- Bank posiada 1000 maszyn
- Czas życia danej wersji oprogramowania wynosi 2 lata
- Każda maszyna obsługuje ok. 200.000 transakcji w czasie cyklu użytkowania jednej wersji oprogramowania
- W sumie ok. 300.000 transakcji bazodanowych dziennie

Przykład specyfikacji niezawodności

Klasa awarii	Przykład	Miara niezawodności
Trwałe, nieniszczące	System nie reaguje na wprowadzenie kolejnej karty. Musi zostać zrestartowany aby zacząć funkcjonować	ROCOF 1 przypadek w ciągu 1000 dni
Przejściowe, nieniszczące	Odczyt kodu z paska magnetycznego dobrej karty jest zakończony niepowodzeniem	POFOD 1 na 1000 transakcji
Trwałe, niszczące	Sekwencja transakcji w sieci powoduje zniszczenie centralnej bazy danych	Niekwalfikowalne!! Nie powinno się to nigdy zdarzyć w trakcie eksploatacji systemu

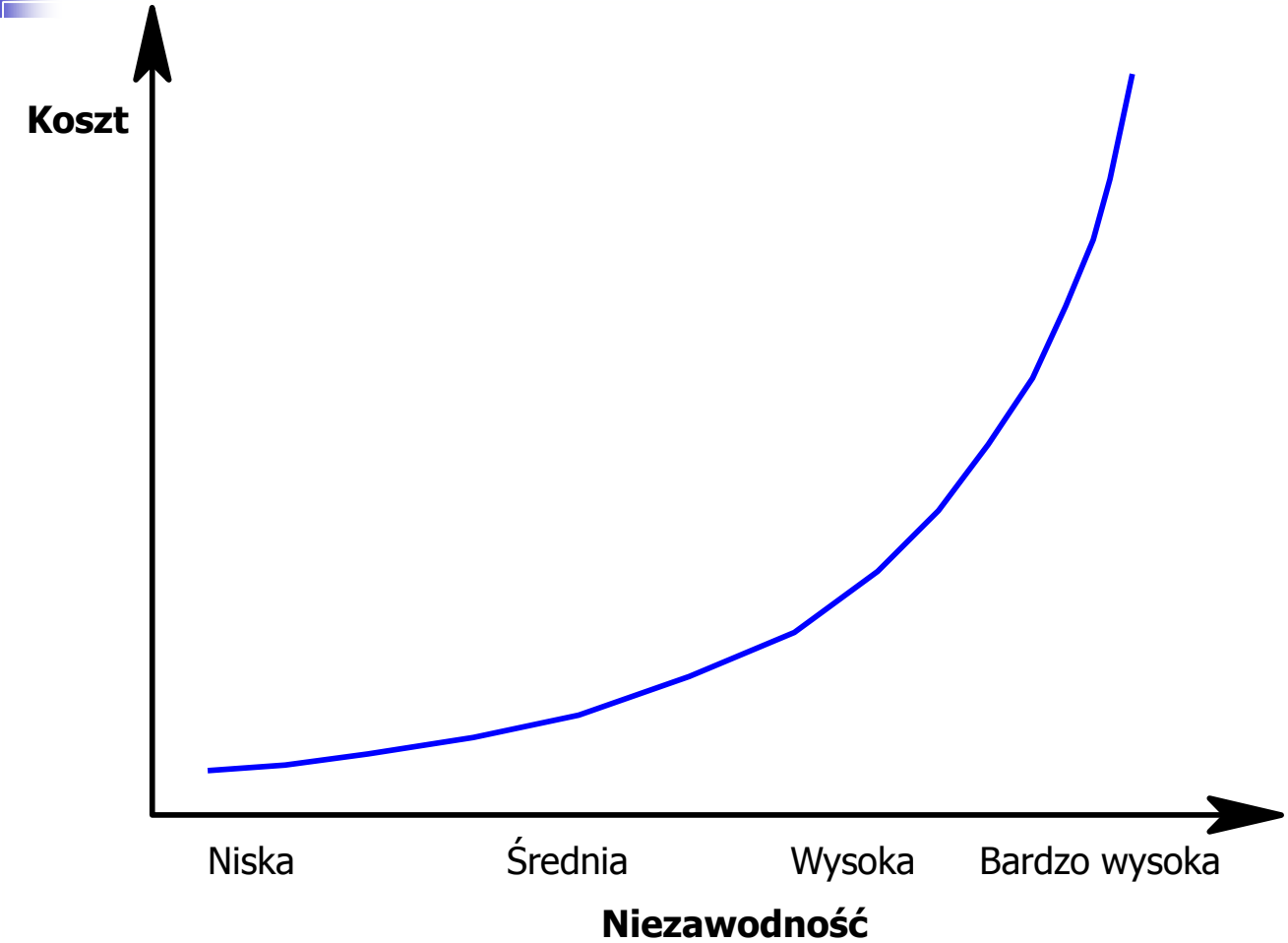
Walidacja specyfikacji

- W praktyce walidacja bardzo restrykcyjnych specyfikacji niezwodności jest niemożliwa!
- Założenie że baza danych nie ulegnie zniszczeniu w czasie działania systemu oznacza miarę POFOD < 1/200.000.000
- Zakładając że symulacja 1 transakcji trwa sekundę, zasymulowanie jednego dnia operacji trwałoby 3.5 dnia
- Całościowy test niezawodności systemu trwałby dłużej niż planowany czas życia systemu!

Niezawodność a czynniki ekonomiczne

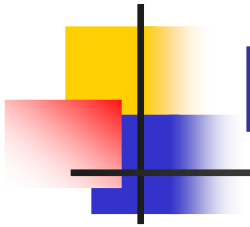
- Z powodu bardzo wysokich kosztów uzyskania wysokiej niezawodności bardziej opłacalne może okazać się zaakceptowanie zawodnego systemu oraz pokrywanie kosztów awarii
  - Ryzyko utraty reputacji => zawodny system
- Dla wielu typów systemów biznesowych wystarczająca może okazać się średnia niezawodność
  - Koszty ew. awarii są minimalne

Koszty rosnącej niezawodności



Testowanie statystyczne

- Testowanie oprogramowania pod kątem niezawodności, nie wykrywania błędów w funkcjonalności
- Wybór danych do testów powinien być zgodny z przewidywanym sposobem używania oprogramowania
- Pomiar liczby błędów pozwala na przewidywanie niezawodności
- Ustala się żądany poziom niezawodności a następnie system jest testowany i poprawiany dopóki nie zostanie osiągnięty zadany poziom



Procedura

- Wyznaczenie wzorca operacyjnego dla oprogramowania
  - Trudne dla nowych systemów
- Generacja zestawu danych odpowiadających wzorcowi
- Wykonanie testów oraz pomiar czasu wykonania pomiędzy awariami
- Pomiar niezawodności po wykonaniu wystarczającej liczby testów (uzyskanie statystyki)

Trudności

- Jak stworzyć wzorzec operacyjny?
  - Szczególnie uwidacznia się dla nowych systemów. Dla już istniejących – baza danych historycznych
- Wysoki koszt uzyskania wzorca
  - Jeśli dane dot. użytkowania systemu były kolekcjonowane (logowane), można je wykorzystać. W przeciwnym wypadku konieczność generacji
    - Trzeba uwzględnić sytuacje nietypowe
- Przy specyfikacji wysokiej niezawodności dochodzą do głosu czynniki statystyczne
  - Trudno wyznaczyć poziom ufności danego wzorca
  - Wzorzec operacyjny może ulec zmianie w czasie

Techniki tworzenia niezawodnego oprogramowania

- Techniki unikania błędów
- Tolerancja błędów, architektury tolerujące błędy
- Obsługa i zarządzanie wyjątkami
- Programowanie defensywne

Postrzeganie niezawodności

- Niezawodność oprogramowania jest jego cechą oczekiwaną przez wszystkich klientów. Dla aplikacji niekrytycznych możliwa jest akceptacja pewnych błędów w oprogramowaniu
  - Jeśli nie są uciążliwe (częste)
  - Nie mają poważnych konsekwencji
- W przypadku aplikacji wymagających wysokiej niezawodności pojawia się potrzeba specjalnych technik programistycznych

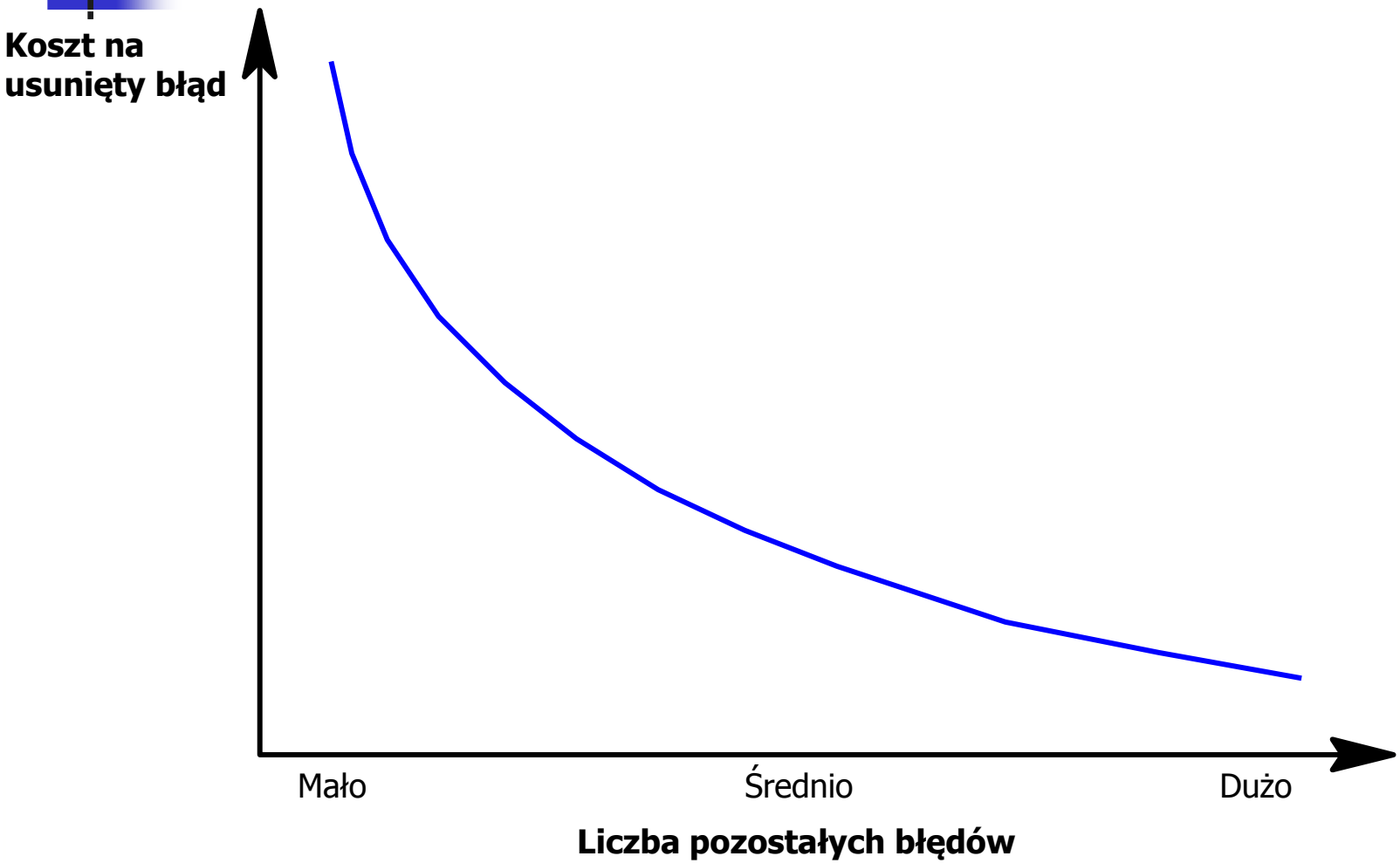
Strategie osiągania niezawodności

- Unikanie błędów
  - Oprogramowanie jest tworzone w taki sposób aby nie zawierało błędów
- Wykrywanie błędów
  - Proces tworzenia oprogramowania jest zorganizowany tak że błędy są wykrywane i poprawiane przed dostarczeniem produktu do klienta
- Tolerancja błędów
  - Oprogramowanie jest stworzone tak że ujawnienie się błędów nie powoduje całkowitego paraliżu systemu

Unikanie błędów

- Obecne metody inżynierii oprogramowania pozwalają na tworzenie oprogramowania praktycznie wolnego od błędów
- Oprogramowanie wolne od błędów oznacza tu oprogramowanie zgodne ze swoją specyfikacją
  - Nie oznacza to że taki system zawsze będzie działał poprawnie
  - Istnieje możliwość wystąpienia błędów w specyfikacji
- Koszt wytworzenia oprogramowania wolnego od błędów jest bardzo wysoki, akceptowalny jedynie dla wąskiej klasy systemów. Często taniej jest zaakceptować istniejące błędy w oprogramowaniu

Koszty usuwania błędów



Tworzenie oprogramowania wolnego od błędów

- Wymagana jest precyzyjna specyfikacja
  - Najlepiej formalna
- W projekcie duży nacisk kładzie się na ukrywanie oraz enkapsulację danych
- Wykorzystywany język oprogramowania musi wspierać kontrolę typów oraz kontrolę w czasie wykonania
- Intensywne wykorzystywanie przeglądów (reviews) na wszystkich etapach procesu
- Organizacja musi posiadać zdefiniowane cele jakości
- Dokładne i szerokie w swoim zakresie testowanie

CMM L 4, 5

Programowanie strukturalne

- Znane od początku lat 70-tych
- Programowanie bez użycia instrukcji goto
  - Wg Dijkstra (1968) konstrukcja generująca błędy
  - Ukształtowało się nawet stwierdzenie: Jakość programisty = 1 / liczba użytych instrukcji goto ☺
- Sterowanie przy pomocy konstrukcji while i if
- Projektowanie metodą top-down
- Odegrało (i odgrywa) ważną rolę ponieważ zapoczątkowało systematyczne podejście do tworzenia oprogramowania

Konstrukcje podatne na błędy (1/2)

- Liczby zmiennoprzecinkowe
  - Nieprecyzyjne ze swojej natury. Brak precyzji może generować błędne skutki porównań
  - Często popełniany błąd: if (a==b) {...}
- Wskaźniki
  - Wskaźnik odnoszący się do niewłaściwego obszaru pamięci może zniszczyć dane. Używanie aliasów utrudnia zrozumienie i modyfikacje programu
- Dynamiczna alokacja pamięci
  - Może powodować przepełnienie jak i wycieki pamięci
- Konstrukcje równoległe
  - Może generować błędy wynikłe z nieprzewidzianych wcześniej interakcji pomiędzy procesami

Konstrukcje podatne na błędy (2/2)

- Rekursja
  - Błędy mogą spowodować przepełnienie pamięci
- Przerwania
  - Wystąpienie przerwania może spowodować zakończenie wykonywania krytycznej operacji.
  - Porównywane z instrukcją goto
- Powyższych konstrukcji NIE powinno się unikać; powinno się je stosować ze szczególną ostrożnością

Ukrywanie informacji

- Informacja powinna być dostępna tylko dla tych części programu które jej potrzebują. Uzyskuje się to poprzez tworzenie obiektów bądź abstrakcyjnych typów danych zawierających dane o stanie oraz udostępniających operacje na tym stanie
- Unika się przez to błędów, gdyż:
  - Zmniejsza się prawdopodobieństwo przypadkowego uszkodzenia danych
  - Dane są otoczone 'warstwą ochronną' a więc błędy w danych mniej się propagują na pozostałe części programu
  - Ponieważ poszczególne dane są umiejscowione w 1 punkcie, zmniejsza się prawdopodobieństwo popełnienia błędu przez programistę a zwiększa – wykrycia błędu przez inspektorów kodu

Typizacja

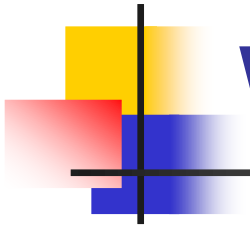
- Podstawą ukrywania informacji jest wykorzystanie języka programowania z typizacją
- Wiele błędów w oprogramowaniu powstaje na skutek przypisywania zmiennym niewłaściwych wartości
  - Język z silną typizacją – kontrola poprawności przypisań w czasie kompilacji
- Nazewnictwo typów zwiększa czytelność programów – modelowanie obiektów świata rzeczywistego
- Deklaracje typów w C++
  - ```
    typedef enum { red, redamber, amber, green}
    TrafficLightColour;
    TrafficLightColour ColourShowing, NextColour;
    ```
    ```

Obiekty oraz abstrakcyjne typy danych

- Implementowane jako obiekty w C++
- Nazwa typu jest deklarowana w obrębie obiektu
- Dozwolone operacje na typie są definiowane jako procedury bądź funkcje
- Reprezentacja typu jest ukryta w części prywatnej
- Ogólne, abstrakcyjne struktury danych mogą być parametryzowane za pomocą nazwy typu
  - Wzorce w C++

Deklaracja kolejki w C++

```
class Queue {
public:
Queue () ;
~Queue () ;
void Put ( int x ) ;
int Remove () ;
int Size( ) ;
private:
int front, back ;
int qvec [100] ;
} ;
```



Wzorce

- Zachowanie obiektów złożonych z innych obiektów jest często takie samo niezależnie od typu obiektów wchodzących w skład
  - Np. kolejka zawsze będzie posiadała operacje typu dodaj, usuń niezależnie od typu poszczególnych elementów
- Wzorce umożliwiają tworzenie ogólnych klas których obiekty mogą być potem tworzone z podaniem konkretnego typu

Deklaracja wzorca kolejki w C++

```
template
<class elem>
class Queue {
public:
Queue ( int size = 100 ) ;
~Queue () ;
void Put ( elem x ) ;
elem Remove ( ) ;
int Size ( ) ;
private:
int front, back ;
elem* qvec ;
} ;
```

Obiekty klasy wzorcowej

- Obiekty klas wzorcowych są „tworzone” w czasie kompilacji tak więc możliwa jest kontrola typów
- C++
  - // Zakładamy że ‘List’ zostało wcześniej zadeklarowane jako typ danych

    Queue <int> Int_queue (50) ;
    Queue <List> List_queue (200) ;

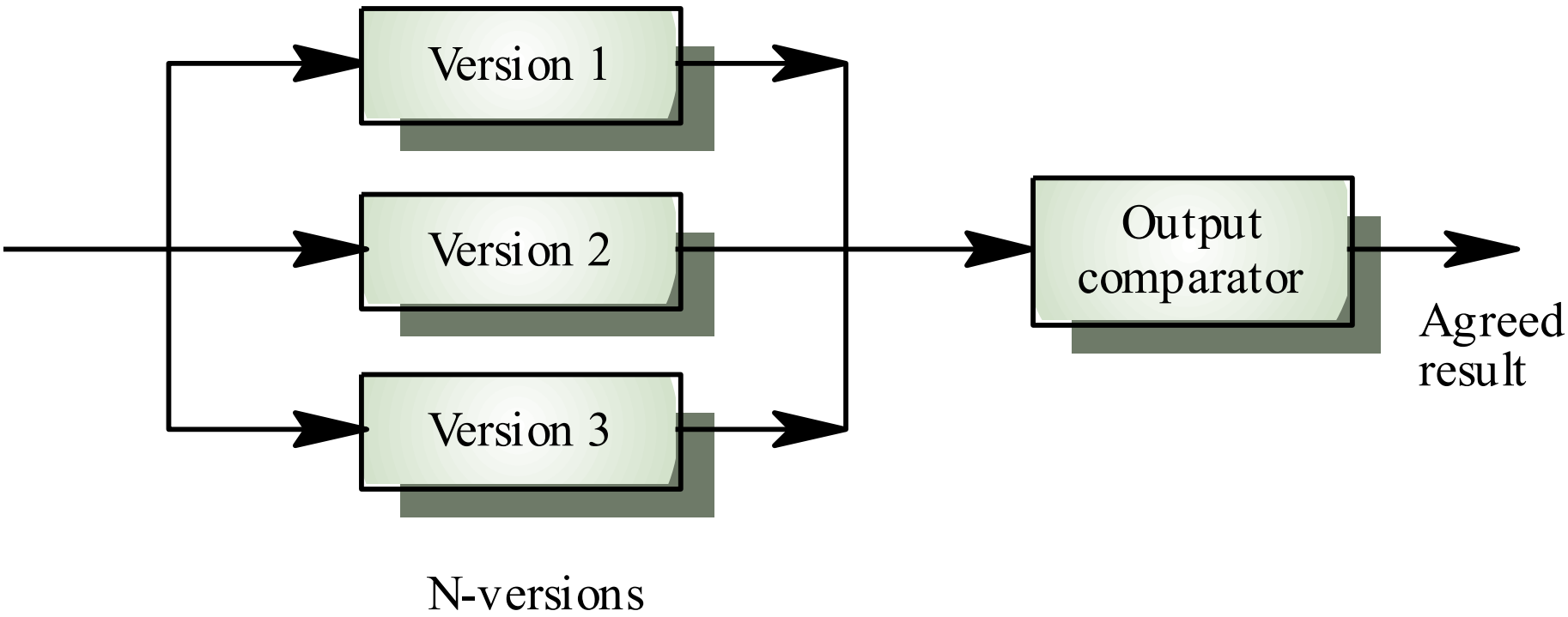
Tolerancja błędów

- W sytuacjach krytycznych system informatyczny musi tolerować ujawnianie się błędów
- Tolerancja błędów oznacza że system potrafi kontynuować swoje działanie pomimo wystąpienia awarii
- Nawet jeśli dany system został zakwalifikowany jako wolny od błędów, powinien on mimo to tolerować błędy
  - Mogą ujawnić się błędy zawarte w specyfikacji
  - Walidacja systemu może okazać się błędna

Tolerancja błędów

- Programowanie w N wersjach (*ang. N-version programming*)
  - Dana specyfikacja jest implementowana w kilku różnych wersjach. Wszystkie wersje danego modułu/funkcji wykonują się jednocześnie dla określonych danych wejściowych a następnie porównuje się uzyskane wyniki. Jest to powszechnie wykorzystywane podejście, np. w samolotach Airbus 320.
  - Nieskuteczne w przypadku błędnej specyfikacji !!
- Odzyskiwalne bloki (*ang. recovery blocks*)
  - Kolejne wersje są wykonywane sekwencyjnie, a uzyskanje wyniki poddawane weryfikacji przez test akceptacyjny. Do dalszego przetwarzania kierowane są dane które przeszły przez test.
  - Problem -> jak stworzyć test akceptacyjny?

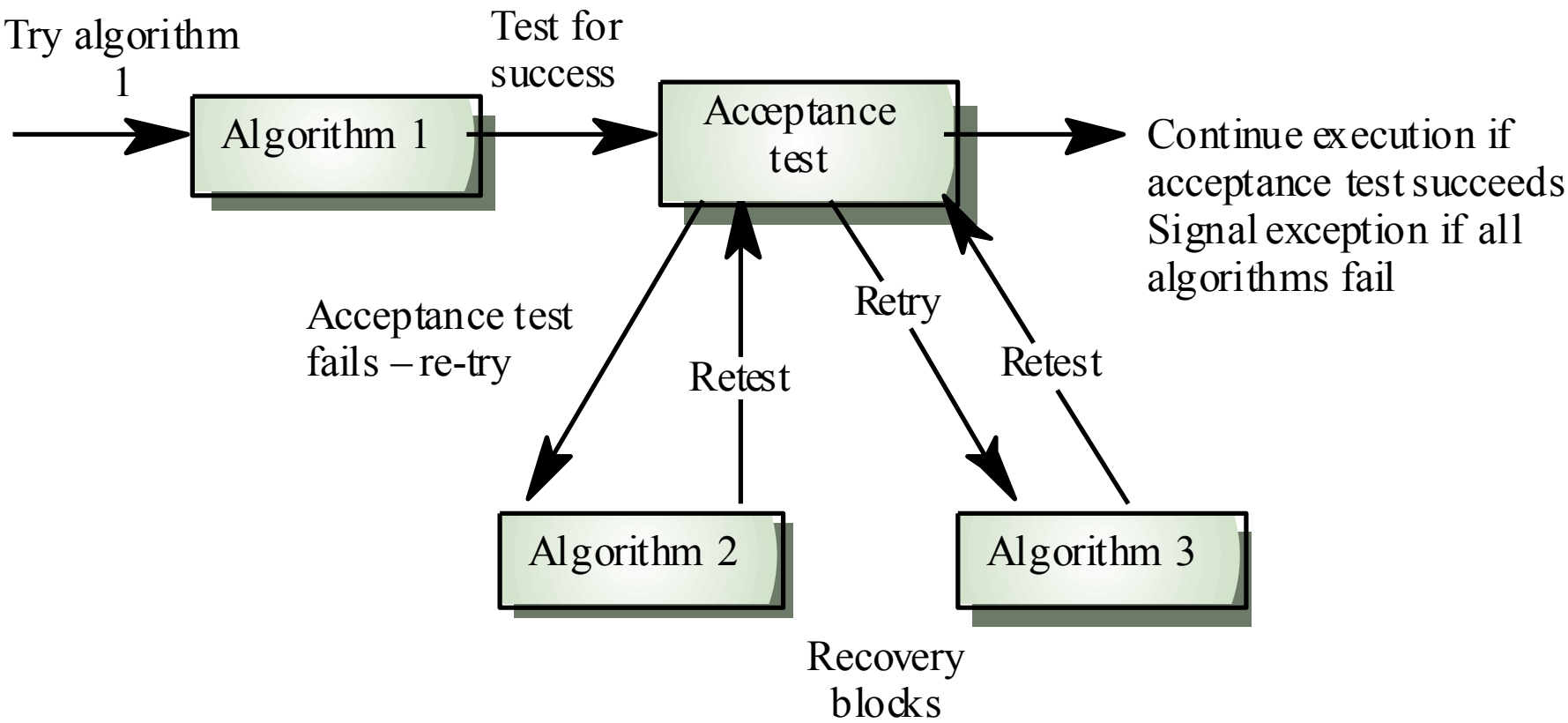
Programowanie w N wersjach



Programowanie w N wersjach - cechy

- Osobne wersje systemu są projektowane i kodowane przez różne zespoły. Zakłada się że istnieje małe prawdopodobieństwo powtórzenia tych samych błędów przez różne zespoły.
- Doświadczenie pokazuje że takie zespoły mają skłonność do (niewłaściwej) interepretacji specyfikacji w podobny sposób oraz do używania podobnych algorytmów

Odzyskiwalne bloki



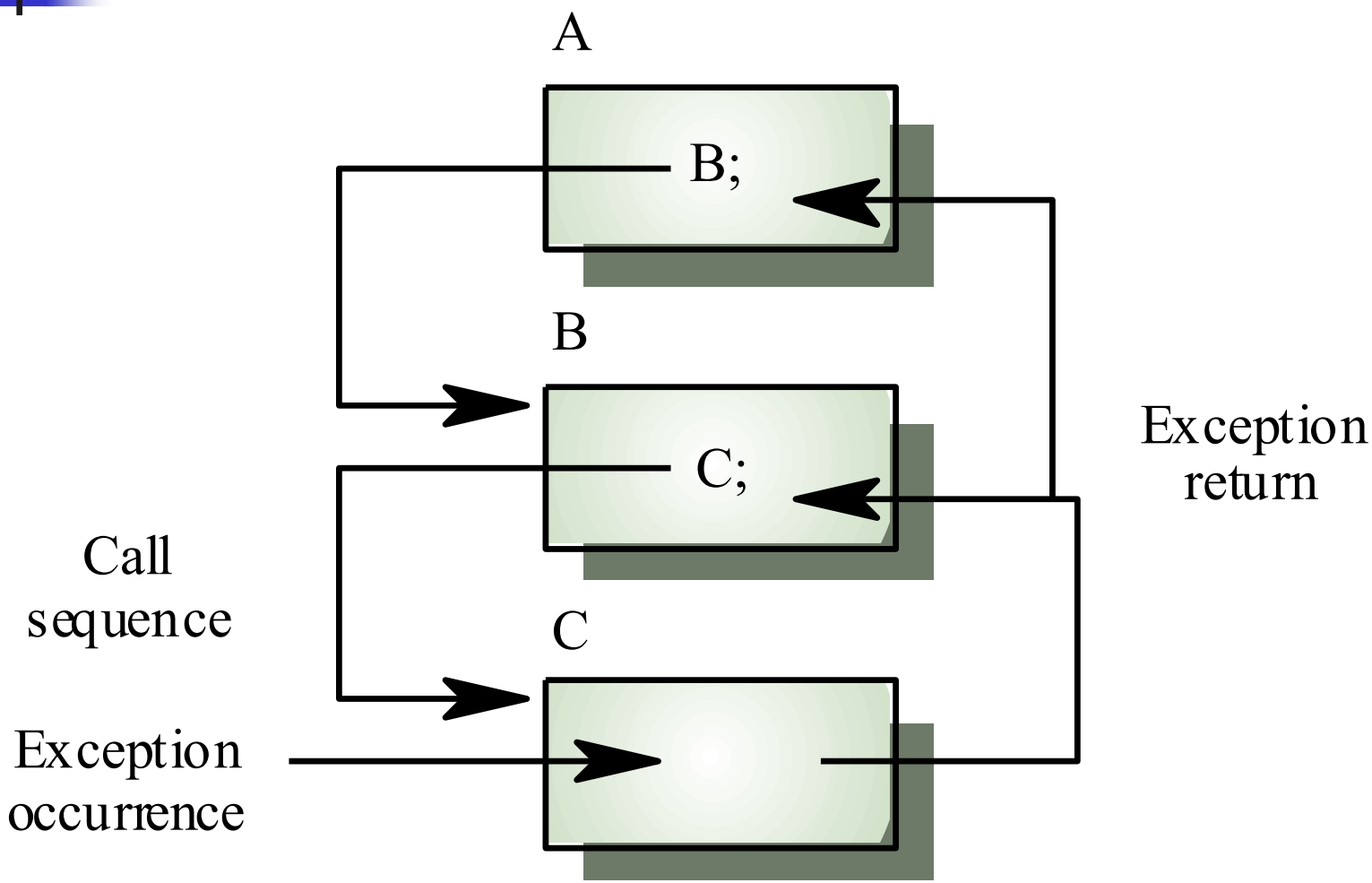
Odzyskiwalne bloki - cechy

- Wymusza użycie różnych algorytmów w różnych wersjach tak więc zmniejsza się prawdopodobieństwo wystąpienia wspólnych błędów
- Stworzenie odpowiedniej procedury testującej jest trudne; musi być niezależne od przyjętej metody obliczeń
- Podobnie jak programowanie w N wersjach, nieodporne na błędy zawarte w specyfikacji

Obsługa wyjątków

- Wyjątek – błąd bądź niespodziewane zdarzenie takie jak np. zanik zasilania
- Kontrukcje obsługujące wyjątki umożliwiają obsługę takich zdarzeń bez konieczności ciągłego sprawdzania statusu i zwracanych przez funkcje wartości
- Wykorzystywanie standardowego sposobu wykrywania wyjątków w przypadku sekwencji zagnieżdżonych wywołań procedur
  - Wymaga dodania szeregu dodatkowych wyrażeń
  - Wprowadza zauważalny narzut czasowy

Wyjątek w zagnieżdżonym wywołaniu procedur



Obsługa wyjątków w C++

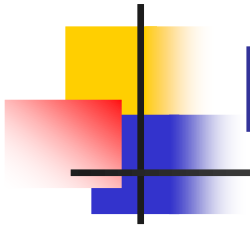
- Słowo kluczowe throw oznacza zgłoszenie wyjątku. Obsługa wyjątku następuje za pomocą instrukcji catch
- Wyjątki są definiowane jako klasy tak więc mogą dziedziczyć swoje właściwości z innych klas wyjątków
- W typowej sytuacji wyjątki są obsługiwane w obrębie bloku w którym zostały zgłoszone
  - Nie propaguje się ich dalej chyba że ich obsługa w danym bloku jest niemożliwa

Przykład: kontroler temperatury

- Steruje urządzeniem chłodniczym utrzymując temperaturę w zadanym zakresie
- Steruje agregatem chłodniczym
- Uruchamia alarm w przypadku przekroczenia maksymalnej dozwolonej temperatury
- Używa zdefiniowanych zewnętrznie obiektów
  - Pump, Temperature_dial, Sensor, Alarm

Kontroler temperatury

```
void Control_freezer ( const float Danger_temp)
{
      float Ambient_temp ;
      // try oznacza że w obrębie bloku będą obsługiwane wyjątki
      try {
            while (true) {
                  Ambient_temp = Sensor.Get_temperature () ;
                  if (Ambient_temp > Temperature_dial.Setting () )
                        if (Pump.Status () == off)
                        {
                              Pump.Switch (on) ;
                              Wait (Cooling_time) ;
                        }
                        else
                        if (Pump.Status () == on)
                              Pump.Switch (off) ;
                  if ( Ambient_temp > Danger_temp )
                        throw Freezer_too_hot ( ) ;
            }  // koniec pętli while
      }  // koniec bloku w którym może nastąpić zgłoszenie wyjątku
      catch ( Freezer_too_hot ) // obsługa wyjątku
            Alarm.Activate () ;
}
```



Podsumowanie (1/2)

- Niezawodność systemu można poprawić stosując techniki unikania błędów oraz tolerancji błędów
- Pewne konstrukcje programistyczne takie jak instrukcja goto, rekursja oraz wskaźniki są ze swojej natury podatne na błędy
- Typizacja danych pozwala na wychwycenie wielu potencjalnych błędów już w czasie kompilacji
- Oprogramowanie tolerujące błędy może kontynuować swoje działanie pomimo obecności błędów

Podsumowanie (2/2)

- Tolerancja błędów wymaga zastosowania
  - Detekcji błędów
  - Oceny zniszczeń (uszkodzeń)
  - Procedur odzyskiwania stanu systemu sprzed awarii
  - Naprawy uszkodzeń
- Programowanie w N wersjach oraz odzyskiwalne bloki to dwa różne podejścia do tolerancji błędów
- Mechanizm obsługi wyjątków może zostać wykorzystany do przywrócenia stabilnego stanu systemu po wystąpieniu jego awarii