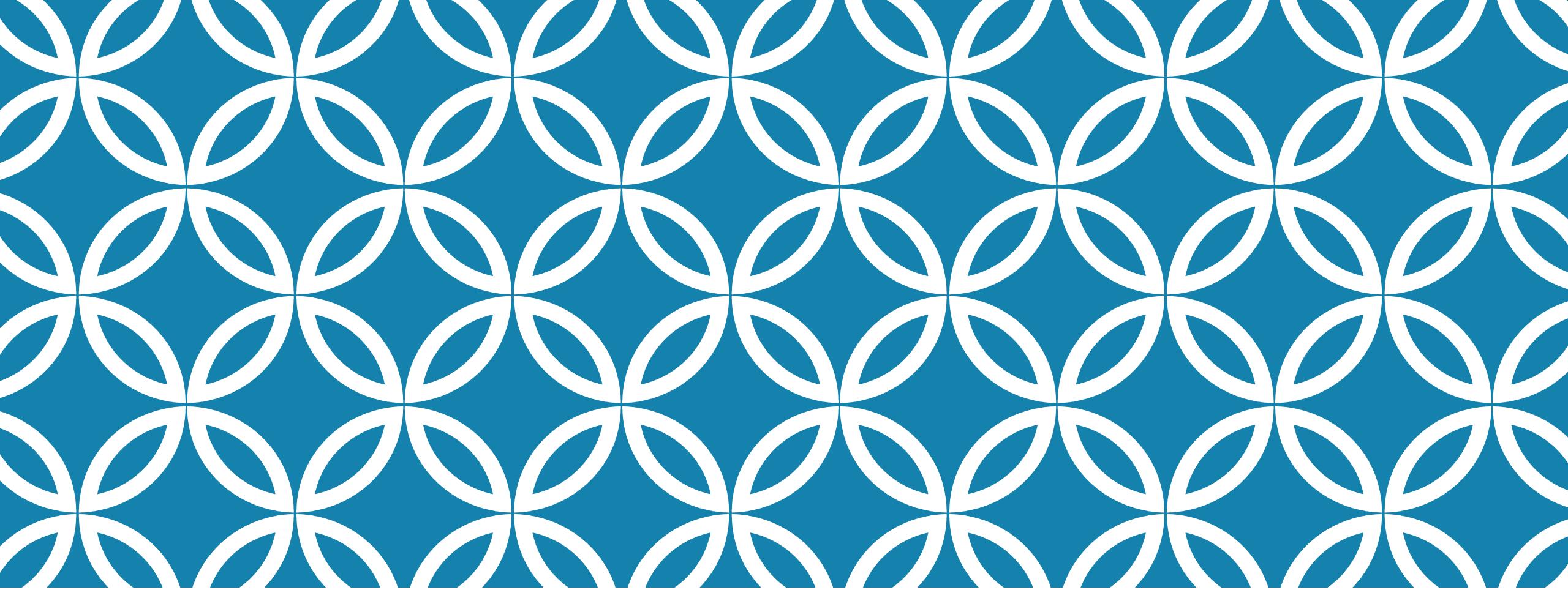


FORMA ZALICZENIA – DWIE MOŻLIWOŚCI

- 1. PEŁNA IMPLEMENTACJA AGENTA SNMP**
- 2. (LUB) Zdecentralizowana Okienkowa Aplikacja działająca na platformie**

GRIDNEOS

- Dowolny atrakcyjny, użyteczny pomysł**
 - Np. (głosowanie, szyfrowana komunikacja, YouTube-podobny interfejs/mechanika do oglądania wyszukiwania filmów itp. ltd.)**
- Technologie: WebRTC, JavaScript, WebGL**
- API zostanie dostarczone**
- Możliwość współpracy przy samym Core**



ASN.1

Oraz Kodowanie BER w kontekście protokołu SNMP

MODELOWANIE PROJEKTOWANIE I
ANALIZA SIECI KOMPUTEROWYCH

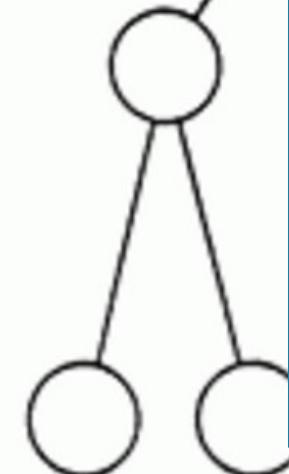
Rafał Skowroński

METODYKA DZIAŁANIA

1. Dowiadujemy się co to SNMP
2. Robimy krótkie zajęcia ćwiczeniowe/wprowadzające w temat by zobaczyć temat w praktyce
3. Projekt składa się z etapów, powiązanych, **funkcjonalnie niezależnych od siebie.**
4. Omawiamy dokładnie każdy etap; dajemy sobie czas na implementację T1 (w między czasie spotykając się), omawiając napotkane problemy.
5. Po upływie czasu T1; sprawdzamy co udało nam się osiągnąć, przydzielamy sobie plusiki; bądź słoneczka
6. Maksymalnie można dostać 3 słoneczka za etap.
7. Każdy tydzień opóźnienia = jedno słoneczko mniej do uzyskania.
8. Słoneczka mają wpływ na ocenę końcową tj. zaliczenie (3.0) od połowy możliwych do uzyskania słoneczek przy założeniu że program ostatecznie działa.



TASK 2:



UWAGA: Maksymalnie można dostać trzy słoneczka.

ISO; OID: 1

- 2) Program daje możliwość zakodowania wartości atomowych jak i złożonych (np. sekwencji zawierającej kolejne sekwencje sekwencji.)
- 3) W każdym przypadku program prosi o podanie OID kodowanego elementu i podanie wartości.
- 4) **Wartość OID może ale nie musi zostać podana.** Przy podaniu OID sprawdzana jest poprawność podawanych danych. Program potrafi zweryfikować poprawność podanej wartości, ORAZ wygenerować kodowanie BER dla danej instancji typu.

MIB-2; OID:
1.3.6.1.2.1

System:
1.3.6.1.2.1.1

sysDescr; OID: 1.3.6.1.2.1.1.1
sysUpTime; OID: 1.3.6.1.2.1.1.3

1) UWAGA: Potrzebujemy móc kodować dowolne zagnieżdżenia typów. Np. sekwencja w sekwencji.

Dla tego powinna też być możliwość rozpoczęcia sekwencji, podania danych wewnętrznych, oraz zakończenia sekwencji.

Alternatywą dla takiego interaktywnego testowania jest zaimplementowanie testów jednostkowych.

ASN.1, SMI, PLIKI MIB *CO TO TAKIEGO? :)*

W skrócie:

Notacja: ASN.1 > SMI **Kodowanie:** BER **Baza danych:** MIB

A konkretniej:

ASN.1 – popularna semantyka / standard używany do opisu struktur danych.

Cel: Reprezentacja danych niezależna od sprzętu.

SNMP (RFC-1157) – definiuje formaty pakietów używanych przez protokół SNMP

SMI (RFC-1155) – definiuje pod-składnie ASN.1 czyli SMI. Jest to składnia której można używać do definiowania obiektów w MIBach.

MIB-II (RFC-1213) – jedna z wielu „grup zadządzalnych” definiujące same obiekty do których mamy dostęp przez sieć (parametry, zmienne, tablice). MIB-II musi być obsługiwany przez każde urządzenie obsługujące protokół SNMP.

FUNKCJONALNOŚĆ: PARSER SMI

CO JEST CELEM ETAPU

Stworzenie klas do przechowywania /reprezentacji danych i wykonywania zadanej w etapie funkcjonalności np. (*to jedynie propozycja*)

- ❖ CMIBParser – klasa odpowiedzialna za parsowanie tekstu i tworząca „drzewo”
- ❖ CDataType – klasa reprezentująca jeden ze sparsowanych typów danych
- ❖ CTrieNode – klasa będąca elementem drzewa (z niej dziedziczymy na elementy przedstawiające liście w drzewie lub nie, jeśli nie to taki element posiadania jedynie nazwe i ID)
- ❖ Lista typów danych (tj obiektów klas CDataType). Zapewniamy powiązanie pomiędzy nimi i liśćiami drzewa tworzonego przez CMIBParser.

PROPAGANDA

Po co to robimy?

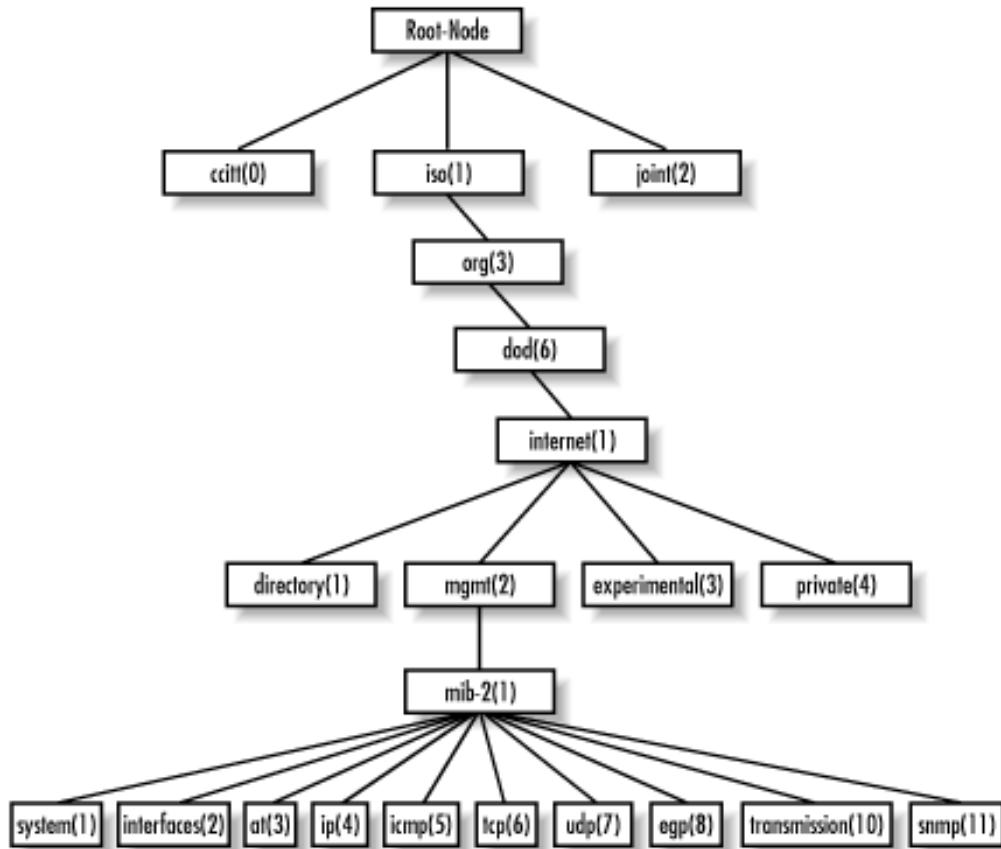
- wszystkie informacje odnośnie typów danych, przydadzą się do sprawdzania czy z sieci otrzymujemy wartości w prawidłowym formacie; w prawidłowym dozwolonym przedziale wartości. Przykład: Do naszego agenta przyjdzie zapytanie SNMP-SET dla wartości sysDescription. Sprawdzimy czy wartość zmiennej jest odpowiedniego typu i czy spełnia wymagania odnośnie długości ciągu.
- informacje odnośnie kodowania w typie zmiennej (APPLICATION, CONTEXT-SPECIFIC itp. Oraz identyfikator typu) będą przydatne w czasie kodowania danych do przesłania w formacie niezależnym od platformy (kodowanie BER)
- Sami sobie dajemy wycisk z wybranego języka programowania
- Wyrażenia regularne, czy też parsowanie ciągów wejściowych jest bardzo często stosowane
- Możemy się poczuć jak twórca parser'a języka programowania (WoW)
- Bo to nie UAM.



STRUKTURA PLIKU MIB-2

```
mib-2      OBJECT IDENTIFIER ::= { mgmt 1 }
system     OBJECT IDENTIFIER ::= { mib-2 1 }
interfaces  OBJECT IDENTIFIER ::= { mib-2 2 }
at         OBJECT IDENTIFIER ::= { mib-2 3 }
ip          OBJECT IDENTIFIER ::= { mib-2 4 }
icmp        OBJECT IDENTIFIER ::= { mib-2 5 }
tcp         OBJECT IDENTIFIER ::= { mib-2 6 }
udp         OBJECT IDENTIFIER ::= { mib-2 7 }
egp         OBJECT IDENTIFIER ::= { mib-2 8 }
transmission OBJECT IDENTIFIER ::= { mib-2 10 }
snmp        OBJECT IDENTIFIER ::= { mib-2 11 }
```

mib-2 jest więc zdefiniowany jako iso.org.dod.internet.mgmt.1, co jest równoznaczne z 1.3.6.1.2.1 w notacji „kropkowej”.



DEFINICJE POCZĄTKOWYCH OBIEKTÓW

W zależności od wersji plików MIB z których zdecydujemy się skorzystać. Obiekty oraz inne identyfikatory mogą być zadeklarowane w odmienny sposób.

-- RFC1155 MIB

RFC1155-SMI DEFINITIONS ::= BEGIN

-- the path to the root

org OBJECT IDENTIFIER ::= { iso 3 }

dod OBJECT IDENTIFIER ::= { org 6 }

internet OBJECT IDENTIFIER ::= { dod 1 }

directory OBJECT IDENTIFIER ::= { internet 1 }

mgmt OBJECT IDENTIFIER ::= { internet 2 }

experimental OBJECT IDENTIFIER ::= { internet 3 }

private OBJECT IDENTIFIER ::= { internet 4 }

enterprises OBJECT IDENTIFIER ::= { private1 }

mib-2 OBJECT IDENTIFIER ::= { mgmt 1 }

END

-- RFC 1902

SNMPv2-SMI DEFINITIONS ::= BEGIN

-- the path to the root

org OBJECT IDENTIFIER ::= { iso 3 }

dod OBJECT IDENTIFIER ::= { org 6 }

internet OBJECT IDENTIFIER ::= { dod 1 }

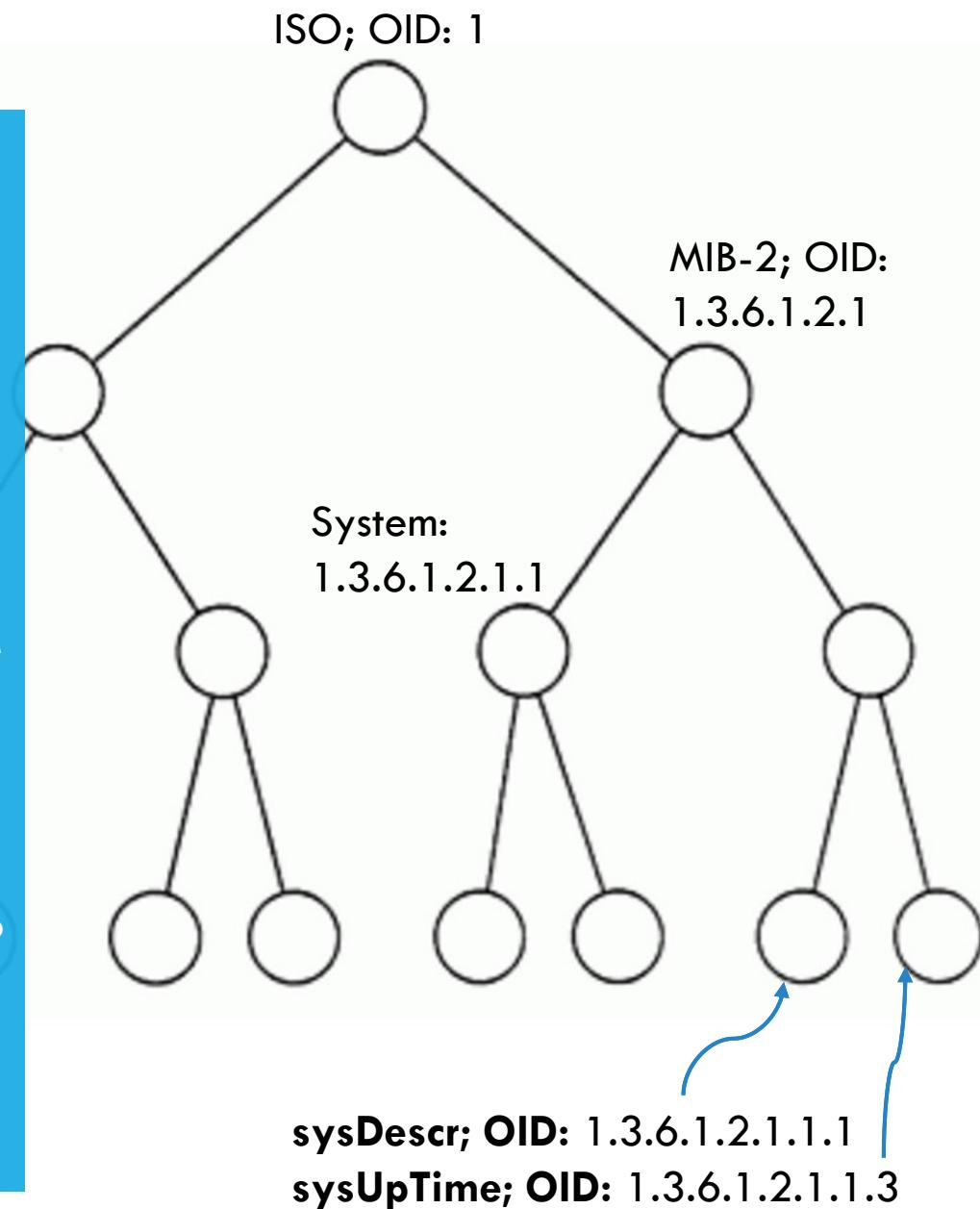
directory OBJECT IDENTIFIER ::= { internet 1 }

mgmt OBJECT IDENTIFIER ::= { internet 2 }

mib-2 OBJECT IDENTIFIER ::= { mgmt 1 }

Obiekty Zarządzalne

1. Każdy „obiekt zarządzalny” w pliku MIB posiada identyfikator OID np. „1.3.6.1.2.1.1.1” oraz nazwę słowną np. sysUpTime.
2. Deklaracja OID'a korzenia (ISO) nie znajduje się w żadnym z plików importów. Potrzebujemy zdefiniować go w implementacji.
3. naturalną formą reprezentacji wszystkich przetworzonych obiektów jest więc „drzewo”.
4. Cały OID reprezentuje ścieżkę od korzenia do danego nod'a. Zamiast całego OID'a nod przechowuje jedynie swój identyfikator tj. jednego integer'a.
5. Zarządzać możemy tylko liśćmi to co powyżej liści stanowi „grupe zarządzalną”.
6. Na drzewo nie trafiają informacje o przetworzonych typach danych. Je możemy przechowywać np. na liście.
7. Każdy nod w drzewie posiada także nazwę tekstową, która może się powtarzać na przestrzeni drzewa. Wyszukując nod'a po nazwie przeszukujemy drzewo „w szerz”. Jeśli przeszukujemy po OID, przechodzimy od korzenia do danego elementu tak jak OID „nakazuje”. Jeśli czegoś nie ma – wyjątek.



LOKALIZACJA PLIKÓW MIB

Program agenta który tworzymy potrzebuje korzystać z plików MIB.

Uznamy, że nasz program korzysta z plików znajdujących się w folderze MIBS pakietu NET-SNMP.

Do testów jako dane wejściowe podajemy plik RFC-1213-MIB.txt

UWAGA: Makra Object

Plik RFC-1213-MIB.txt będzie importować
inne pliki.

WYRAŻENIA REGULARNE

„Wyrażeń regularnych” możemy używać do odnajdowania ciągu znaków tekście spełniających warunki zawarte we wzorcu.

Potrzebujemy spreparować wyrażenia regularne opisujące elementy które będziemy wyszukiwać, czy też – rozpoznawać, w plikach MIB.

[Więcej informacji](#)

Proponuję preferować zagnieżdżone wyrażenia regularne tj. przy użycia dyrektyw warunkowych IF, zamiast korzystać z bardzo skomplikowanych wyrażeń regularnych.

W naszych zastosowaniach szczególnie przydatne może okazać się „wyszukiwanie w przód” oraz opcjonalne grupy (np. w przypadku parsowania typów danych gdzie niektóre pola są opcjonalne).

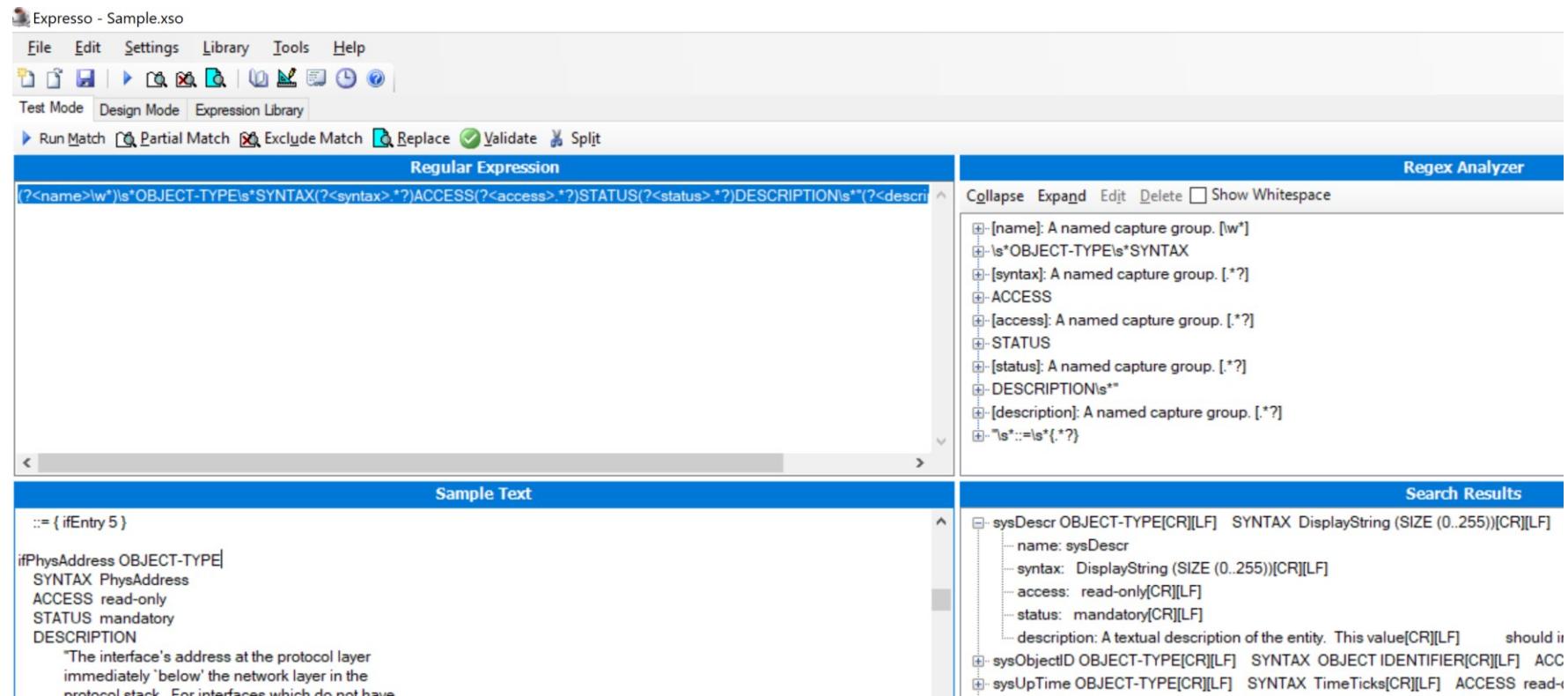
C++11 UWAGA:

C++11 posiada wsparcie dla opcjonalnych grup.
Występuje problem z grupami nazwanymi, można się do nich odnieść po ich pozycji.

WYRAŻENIA REGULARNE

Do parsowania plików wejściowych będzie nam potrzebny silnik wyrażeń regularnych. Np. REGEX.

Do testowania/projektowania wyrażeń regularnych możemy wykorzystać darmowy program EXPRESSO: [Pobierz](#)



WYRAŻENIA REGULARNE C.D

Przykładowy (nie idealny) REGEX do parsowania makr OBJECT-TYPE:

`\w*\s*OBJECT-TYPE\s*SYNTAX.*?ACCESS.*?STATUS.*?DESCRIPTION\s*".*?"\s*::=\s*{.*?}`

Na przykładzie platformy .NET aby zadziałał poprawnie należy zaznaczyć flagę SINGLE-LINE, przydatna też będzie flaga COMPILED.

W przykładzie:

\w* – dowolna ilość znaków alfanumerycznych – łapie nazwę obiektu

\s* – dowolna ilość znaków pustych UWAGA: flaga SINGLE-LINE sprawia że łapie też znaki nowej linii

.*? - – dowolna ilość dowolnych znaków UWAGA: ? Powoduje że działa w trybie **NON-GREEDY** (*nie zachłannym*, czyli nie złapie wszystkich znaków do końca ciągu wejściowego tylko sprawdza jeszcze co jest dalej; można nazwać też przeszukiwaniem z wyprzedzeniem bo patrzymy ,oknem przeszukiwania' co jeszcze jest dalej w wyrażeniu regularnym)

WYRAŻENIA REGULARNE C.D

Przydatne mogą się okazać „GRUPY”. Przykładowo, silnik REGEX na platformie .NET wspiera nazwane grupy (*named groups*). Mogą być pomocne przy odczytywania fragmentów z rozpoznanego ciągu danych. Za pomocą *grup nazwanych* przeróbmy wcześniejsze wyrażenie aby w łatwy sposób pobrać poszczególne pola:

```
(?<name>\w*)\s*OBJECT-TYPE\s*SYNTAX(?<syntax>.*?)ACCESS(?<access>.*?)STATUS(?<status>.*?)DESCRIPTION\s*" (?<description>.*?)"\s*::=\s*{.*?}
```

REZULTAT W CZASIE TESTÓW (program EXPRESSO):

Search Results

- sysDescr OBJECT-TYPE[CR][LF] SYNTAX DisplayString (SIZE (0..255))[CR][LF] ACCESS read-only
 name: sysDescr
 syntax: DisplayString (SIZE (0..255))[CR][LF]
 access: read-only[CR][LF]
 status: mandatory[CR][LF]
 description: A textual description of the entity. This value[CR][LF] should include the full na

CO ZNAJDZIEMY W MIBACH

RFC1213-MIB DEFINITIONS ::= BEGIN

IMPORTS

```
mgmt, NetworkAddress, IpAddress, Counter, Gauge,  
TimeTicks  
FROM RFC1155-SMI  
OBJECT-TYPE ←  
FROM RFC-1212;
```

Importy

Do „czarnej listy” –
nie przetwarzamy

-- This MIB module uses the extended OBJECT-TYPE macro as
-- defined in [14];

-- MIB-II (same prefix as MIB-I)

mib-2 OBJECT IDENTIFIER ::= { mgmt 1 }

-- textual conventions

DisplayString ::=
OCTET STRING

-- This data type is used to model textual information taken
-- from the NVT ASCII character set. By convention, objects
-- with this syntax are declared as having

Uwaga: może być też zapis w formacie:
iso(1) org(3) dod(6) internet(1) mgmt(2) mib-2(1)
system(1) 3 // z ID w nawiasie przy nazwie

-- groups in MIB-II

system OBJECT IDENTIFIER ::= { mib-2 1 }

interfaces OBJECT IDENTIFIER ::= { mib-2 2 }

at OBJECT IDENTIFIER ::= { mib-2 3 }

ip OBJECT IDENTIFIER ::= { mib-2 4 }

Deklaracje samych identyfikatorów

Deklaracje obiektów

sysUpTime OBJECT-TYPE

SYNTAX TimeTicks

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The time (in hundredths of a second) since the
network management portion of the system was last
re-initialized."

::= { system 3 }

Deklaracje typów danych (uwaga, mogą wystąpić w samych OBJECT TYPE)

IpAddress ::=

[APPLICATION 0] -- in network-byte order
IMPLICIT OCTET STRING (SIZE (4))

Counter ::=

[APPLICATION 1]
IMPLICIT INTEGER (0..4294967295)

Gauge ::=

[APPLICATION 2]
IMPLICIT INTEGER (0..4294967295)

TimeTicks ::=

[APPLICATION 3]
IMPLICIT INTEGER (0..4294967295)

CO ZNAJDZIEMY W IMPORTACH: WSZYSTKO😊

IMPORTS

```
mgmt, NetworkAddress, IpAddress, Counter, Gauge,  
TimeTicks  
FROM RFC1155-SMI  
OBJECT-TYPE  
FROM RFC-1212;
```

Do „czarnej listy” –
nie przetwarzamy.
Nie dysponujemy
pełnym leksytem
składni ASN.1

Propozycja: Zignorujmy wyliczenia tego co ma być
zimportowane; importujmy wszystko po napotkaniu pliku do
zimportowania.

TYPY DANYCH

W protokole SNMP mamy do czynienia z pewnym zbiorem przesyłanych oraz przetwarzanych typów danych. Podstawowe typy danych są określone w ramach SMI, która to notocja jest podzbiorem ASN.1

DOZWOLONE TYPY DANYCH

SMI obsługuje „prymitywne” typy, ze składni ASN.1 czyli:

1. INTEGER
2. OCTET STRING
3. OBJECT IDENTIFIER
4. NULL



W MiBach, za pomocą podstawowych „prymitywnych” typów danych możemy definiować bardziej złożone typy.

Oraz jeden typ złożony - [SEKWENCJE](#)

TYP *SEQUENCE (OF)*

Cel:

Można przyporównać do struktury w C (może zawierać wiele typów)

Składnia

SEQUENCE { <type1>, ..., <typeN> } – może zawierać elementy różnego typu

Uwagi:

W porównaniu z notacją ASN.1, SMI nie pozwala na użycie parametru OPTIONAL oraz DEFAULT dla parametrów. Po zdefiniowaniu wszystkie muszą być podane.

SKŁADNIA:

SEQUENCE OF <entry> - może zawierać elementy jedynie jednego typu
(tak jak tablica w C)

PROPOZYCJA: Aby obsługiwać sekwencje możemy np. stworzyć uniwersalną klasę **TypDanych** w której stworzymy listę zawierającą typów które sekwencja zawiera.

TYPY DZIEDZICZONE Z TYPÓW PODSTAWOWYCH

Dodatkowe typy danych mogą być zadeklarowane w dowolnym z zainportowanych plików.

- Składają się z podstawowych typów danych ASN.1

PRZYKŁADY:

IpAddress ::=
[APPLICATION 0]
IMPLICIT OCTET STRING (SIZE (4)) ← Adres IP ma 32 bity = 4 bajty – długość ciągu bajtów

Counter ::=
[APPLICATION 1]
IMPLICIT INTEGER (0..4294967295) ← Dozwolone wartości z przedziału 0 - 4294967295

Gauge ::=
[APPLICATION 2]
IMPLICIT INTEGER (0..4294967295)

TimeTicks ::=
[APPLICATION 3]
IMPLICIT INTEGER (0..4294967295)



UWAGA: Potrzebujemy zawsze pamiętać jaki jest typ bazowy.
Tj. czy mamy do czynienia z liczbą czy ciągiem bajtów.

PARSOWANIE NOWYCH TYPÓW DANYCH

IpAddress ::=

[APPLICATION 0] -- in network-byte order
IMPLICIT OCTET STRING (SIZE (4))

Pola Opcjonalne

Wykorzystujemy wyrażenie regularne wyłapujące bloki zawierające:



1) ciąg znaków alfa-numerycznych

2) znak przypisania ::=

2) nawiasy prostokątne, w środku:

- Jedna z czterech wartości: UNIVERSAL, APPLICATION, CONTEXT-SPECIFIC, PRIVATE – wartość ta oznacza „widoczność” deklarowanego typu.

- LICZBA (w tym przypadku 0) oznaczająca identyfikator typu

3) Słowo kluczowe IMPLICIT lub EXPLICIT

4) Nazwa typu macierzystego (w tym przypadku OCTET STRING)

5) OPCJONALNE rozmiar danych / dozwolone wartości. Obsługujemy dwie notacje ograniczeń:

- notacja (SIZE(**ROZMIAR_STRUKTURY**)) - w przypadku stałej długości

- notacja (SIZE(**ROZMIAR_MIN, ROZMIAR_MAX**)) – dozwolone rozmiary

- notacja (**WARTOSC_MINIMALNA..WARTOSC_MAX**)

Znaki puste (white space):
spacje, tabulacje, znaki nowej linii

Obsługujemy jedynie takie format zapisu. W wyniku czego może się zdarzyć, że OBJECT TYPE zawiera nieobsługiwany przez nas typ danych.

PRZYKŁADOWY REGEX (DLA PARSOWANIA TYPÓW)

Poniżej przykładowe wyrażenie regularne parsujące nowe typy danych z wykorzystaniem *named capture groups* do rozpoznania poszczególnych pól:

| w*|s*::=|s*|/[s*|w*|s*/?<typeID>|d+)|s*|]/|s*|w+|s*/?<parentType>|w+|s*|w*)|s*/?<restrictions>|/?.*?\\|\\)?



Przykładowy efekt:

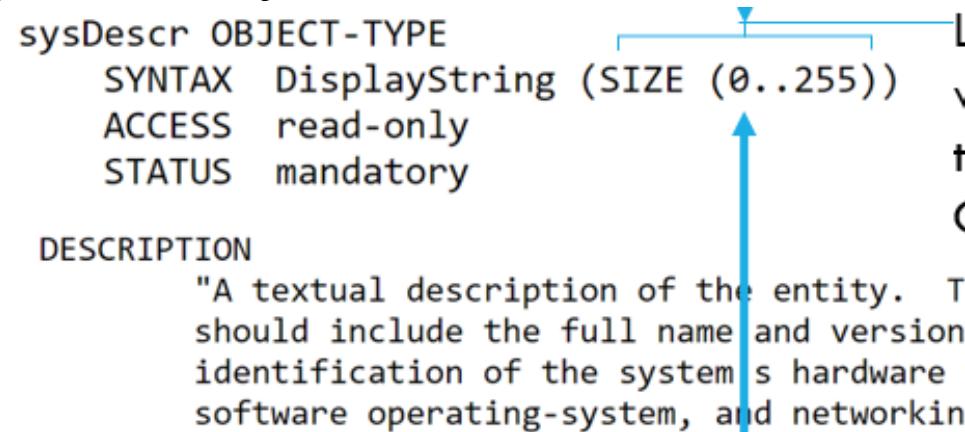
Przykładowe wyrażenie może zawierać błędy / i zawiera braki.

Sample Text	Search Results
ipAddress ::= [APPLICATION 0] IMPLICIT OCTET STRING (SIZE(4))	ipAddress ::=[CR][LF] [APPLICATION 0][CR][LF]IMPLICIT OCTET STRING (SIZE(4)) typeID: 0 parentType: OCTET STRING restrictions: (SIZE(4))
ipAddress2 ::= [APPLICATION 1] IMPLICIT OCTET STRING (SIZE(4)) fdfg	ipAddress2 ::=[CR][LF] [APPLICATION 1][CR][LF]IMPLICIT OCTET STRING (SIZE(4)) typeID: 1 parentType: OCTET STRING restrictions: (SIZE(4))
valve ::= [APPLICATION 33] IMPLICIT INTEGER (0..255) fdfg	valve ::=[CR][LF] [APPLICATION 33][CR][LF]IMPLICIT INTEGER (0..255) typeID: 33 parentType: INTEGER restrictions: (0..255)

GDZIE MOGĄ SIĘ POJAWIĆ, GDZIE PRZECHOWYWAĆ

- Typy danych mogą pojawić się osobno; jak i w deklaracji makra OBJECT TYPE:

```
sysDescr OBJECT-TYPE
    SYNTAX  DisplayString (SIZE (0..255))
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "A textual description of the entity. T
         should include the full name and version
         identification of the system's hardware
         software operating-system, and networkin
```



- **Propozycja:** Przechowujmy obiekty typów danych poza drzewem => nie mają identyfikatorów. Wiele obiektów w drzewie może się odnosić do tych samych typów danych.
- **Propozycja:** Po napotkaniu elementu w makrze OBJECT-TYPE, utwórzmy dynamicznie nowy typ danych, o nazwie sysDescr23432 i dodajmy do LISTY typów. Plus wskaźnik na ten typ w obiekcie sysDescr.

MAKRO OBJECT-TYPE

- 1) Plik MIB składa się z wielu obiektów **OBJECT-TYPE**
- 2) Każdy taki obiekt deklaruje parametr do **odczytu/zapisu** wraz z jego typem danych, identyfikatorem OID, prawami dostępu oraz opisem.
- 3) One reprezentują właśnie ‚obiekty zarządzalne’

Każdy obiekt zadeklarowany w pliku MIB można przymocować do interfejsów w językach programowania wysokopoziomowego takich jak C++ / C#. MIB sam w sobie nie zawiera wartości. Definiuje jedynie zasady i to do czego mamy dostęp.

ZAPISUJEMY WSZYSTKO DOTYCZĄCE OBIEKTU

```
sysUpTime OBJECT-TYPE
SYNTAX  TimeTicks
ACCESS  read-only
STATUS   mandatory
DESCRIPTION
        "The time (in hundredths of a second) since the
        network management portion of the system was last
        re-initialized."
 ::= { system 3 }

sysContact OBJECT-TYPE
SYNTAX  DisplayString (SIZE (0..255))
ACCESS  read-write
STATUS   mandatory
DESCRIPTION
        "The textual identification of the contact person
        for this managed node, together with information
        on how to contact this person."
 ::= { system 4 }
```

Tj. W przypadku OBJECT TYPE Zapisujemy:

- TYP-DANYCH
- Uprawnienia
- OPIS
- Status

PROPOZYCJA: W docelowej dopasowanej klasie/strukturze.

Dane przechowujmy w efektywnym formacie:

- 'Enum' dla wartości pola ACCESS
- std::string dla wartości pola DESCRIPTION itp. Itd.

Przechowywanie zbędnych stringów nie jest dobrym
pomysłem😊 (czas porównywania, konwersji, miejsce w
pamięci)

DOSTĘP DO DANYCH W DRZEWIE

- Potrzebujemy móc odpytywać elementy w drzewie o pojedyncze wartości. Tj
 - Podaj mi swój ID? Podaj jaki masz DESCRIPTION? Powiedz czy można ciebie zapisywać?
 - Jaki masz typ danych?
- Po otrzymaniu informacji o typie danych, potrzebujemy móc znowuż odpytywać typ danych o informacje. Tj.
 - Powiedz jaki masz typ bazowy? (liczba czy ciąg bajtów)? Powiedz z jakiego przedziału wartości są dozwolone?
- Potrzebne jest więc powiązanie obiektów w drzewie z obiektami reprezentującymi typy danych. Jak to zrobić? Po nazwie i przeszukiwać listę obiektów typów przy każdym zapytaniu? Wskaźniki w elementach drzewa wskazujące na obiekty typów

PARSOWANIE IDENTYFIKATORÓW OID

Deklaracje identyfikatorów OID zawierają się w klamrach. Mogą zawierać:

- Nazwę słowną np. iso
- Nazwę słowną wraz z liczbą np. mgmt(2)
- Liczbe np. 1

Przykłady:

```
mib-2      OBJECT IDENTIFIER ::= { mgmt 1 }  
internet   OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
```

Postępujemy w następujący sposób:

- 1) Jeśli napotkamy na nazwę słowną np. mgmt bez nawiasu – szukamy całego OID w drzewie, następujące później elementy będą dziećmi. Nie będzie sytuacji: ~~iso mgmt dod~~
- 2) Jeśli napotkamy na nazwę słowną z nawiasem np. org(3) oznacza liścia org z identyfikatorem 3 może ich być kilka po sobie. Org(3) dod(6) – oznacza liścia dod z identyfikatorem 6 pod rodzicem Org z identyfikatorem 3
- 3) Na końcu zawsze będzie liczba – jest potrzebny identyfikator dla nowego elementu

PARSOWANIE MAKR OBJECT TYPE

```
sysDescr OBJECT-TYPE
    SYNTAX  DisplayString (SIZE (0..255))
    ACCESS  read-only
    STATUS  mandatory
```

```
DESCRIPTION
    "A textual description of the entity. This value
     should include the full name and version
     identification of the system's hardware type,
     software operating-system, and networking
     software. It is mandatory that this only contain
     printable ASCII characters."
 ::= { system 1 }
```

Ograniczenia mogą znajdować się w deklaracji typu, ale mogę też znajdować się w deklaracji obiektu. W takim przypadku ograniczenia w deklaracji obiektu mają większy priorytet.

Lub format ograniczeń (`MIN_VAL.. MAX_VAL`) – identycznie jak w przypadku ograniczeń przy deklaracjach typów można wykorzystać te same wyrażenia regularne.

Czyli ograniczenia potrzebujemy przechowywać w 2 miejscach:

- W typie danych
- W danym nodzie drzewa

LUB

Tworzymy dynamicznie dodatkowe typy danych

Makro OBJECT-TYPE – „hard-kodujemy” tzn. tworzymy wyrażenie regularne, które pozwala na wyłuskanie potrzebnych elementów. Elementy występują zawsze w tej samej kolejności. Tj. słowa kluczowe OBJECT-TYPE, SYNTAX, ACCESS, STATUS, DESCRIPTION, oraz następujący na końcu OID. Inne podejście wymagałoby dynamicznej obsługi notacji tworzącej dowolne makra w notacji ASN.1

Dla cętnych: obsługa pola DEFVAL – zawierającego wartość domyślną dla danej zmiennej. [RFC](#) punkt 4.1.7 opisuje możliwe formaty domyślnych wartości.

PARSOWANIE MAKR OBJECT TYPE

```
sysDescr OBJECT-TYPE
    SYNTAX DisplayString (SIZE (0..255))
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "A textual description of the entity. This value
        should include the full name and version
        identification of the system's hardware type,
        software operating-system, and networking
        software. It is mandatory that this only contain
        printable ASCII characters."
    ::= { system 1 }
```

Ograniczenia mogą znajdować się w deklaracji typu, ale mogę też znajdować się w deklaracji obiektu. W takim przypadku ograniczenia w deklaracji obiektu mają większy priorytet.

Lub format ograniczeń (`MIN_VAL.. MAX_VAL`) – identycznie jak w przypadku ograniczeń przy deklaracjach typów można wykorzystać te same wyrażenia regularne.

Czyli ograniczenia potrzebujemy przechowywać w 2 miejscach:

- W typie danych
- W danym nodzie drzewa

LUB

Tworzymy dynamicznie dodatkowe typy danych

Wynika stąd że właściwości / ograniczenia danego typu danych mogą być zdefiniowane

- 1) Za pomocą deklaracji nowego typu danych tj. opisano wcześniej *LUB*
- 2) Podając ograniczanie w tym miejscu bezpośrednio w deklaracji OBJECT-TYPE'a

W efekcie w tym miejscu tworzy się nowy typ danych wykorzystywany wyłącznie przez sysDescr. Jak do tego podejść? Możemy dodać nowy typ danych do naszej listy typów i odniesienie do niego w obiekcie reprezentującym sysDescr.

OBIEKT SMI C.D

Przykładowe obiekty z MIB-II wykorzystujące
już te makro:

```
sysUpTime OBJECT-TYPE
    SYNTAX TimeTicks
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "The time (in hundredths of a second) since the
         network management portion of the system was last
         re-initialized."
    ::= { system 3 }
```

```
sysContact OBJECT-TYPE
    SYNTAX DisplayString (SIZE (0..255))
    ACCESS read-write
    STATUS mandatory
    DESCRIPTION
        "The textual identification of the contact person
         for this managed node, together with information
         on how to contact this person."
    ::= { system 4 }
```



Nie piszemy pełnego parsera ASN.1; zajmujemy się tylko składnią SMI.
Czyli np. projektujemy REGEX'a do obsługi składni OBJECT-TYPE:

Fragment RFC 1155-SMI zawierający definicje marka OBJECT TYPE:

```
ObjectName ::= OBJECT IDENTIFIER
OBJECT-TYPE MACRO ::=
BEGIN
    TYPE NOTATION ::= "SYNTAX" type (TYPE ObjectSyntax)
                    "ACCESS" Access
                    "STATUS" Status
    VALUE NOTATION ::= value (VALUE ObjectName)

    Access ::= "read-only"
              | "read-write"
              | "write-only"
              | "not-accessible"
    Status ::= "mandatory"
              | "optional"
              | "obsolete"
END
```

SZKIC LOGIKI DODAWANIA ELEMENTU DO DRZEWA

C++, Na przykładzie obiektu OBJECT IDENTIFIER

```
void CTree::addObjectIdentifier (void * currentNode, std::vector<CIDNamePair> identifiers)
{
bool hasPath=false; uint32_t foundAt=0;

for(auto child&: currentNode->getChildren()) //iterujemy po wszystkich ,dzieciach' bieżącego nod'a
    {
        if(child->getID() == identifiers[0]->getID()) //jeśli znaleźliśmy obiekt którego ID pokrywa z bieżącym fragmentem OID'a
        {
            hasPath=true;
            break;//przerywamy przeszukiwanie
        }
        foundAt++;
    }

    if(hasPath) //jeśli odnaleźliśmy istniejącego nod'a pod tym indeksem
    {
        OID.erase(OID.begin());// usuwamy pierwszą parę ID/nazwa- została odnaleziona.
        addObjectIdentifier(children[i],OID);// wchodzimy do środka
    }
    else{ currentNode->addChildObjectID(identifiers[0]->getID(), identifiers[0]->getName());
        addObjectIdentifier(currentNode->getChildAt(identifiers[0]->getID(), identifiers));
    }
}
```

Rozwiążanie oparte na rekurencji. W jaki sposób wyglądałoby rozwiążanie je pomijające? Na Co jest podatne rozwiążanie rekurencyjne? (przepelnienie stosu?)

1. Wskaźnik VOID – mało bezpieczny; brak informacji o typie w czasie pisania kodu. Używajmy smart-pointerów?

CZY TA FUNKCJA JEST IDEALNA? *NIE* 😊

```
void CTree::addObjectIdentifier (void * currentNode, std::vector<CIDNamePair> identifiers)
{
    bool hasPath=false; uint32_t foundAt=0;

    for(auto child&: currentNode->getChildren()) //iterujemy po wszystkich „dzieciach” bieżącego nod'a
    {
        if(child->getID() == identifiers[0]->getID()) //jeśli znaleźliśmy obiekt którego ID pokrywa z bieżącym fragmentem OID'a
        {
            hasPath=true;
            break;//przerywamy przeszukiwanie
        }
        foundAt++;
    }

    if(hasPath) //jeśli odnaleźliśmy istniejącego nod'a pod tym indeksem
    {
        OID.erase(OID.begin()); // usuwamy pierwszą parę ID/nazwa- została odnaleziona.
        addObjectIdentifier(children[i],OID); // wchodzimy do środka
    }
    else{ currentNode->addChildObjectID(identifiers[0]->getID(), identifiers[0]->getName());
        addObjectIdentifier(currentNode->getChildAt(identifiers[0]->getID(), identifiers));
    }
}
```

2. Wektor, wolno obsługuje usuwanie czy dodawanie elementów z przodu. (pamięć jest alokowana/de-alokowana/kopiowana)

5. Brak kontroli danych. Co jeśli `OID.size()==0` ?

4. Brak obsługi wyjątków.

5. Pokrewna logika rozmyta pomiędzy CTree a same Nody

3. Brak zwracanej informacji o powodzeniu operacji.

KOLEJNOŚĆ PRZETWARZANIA ELEMENTÓW

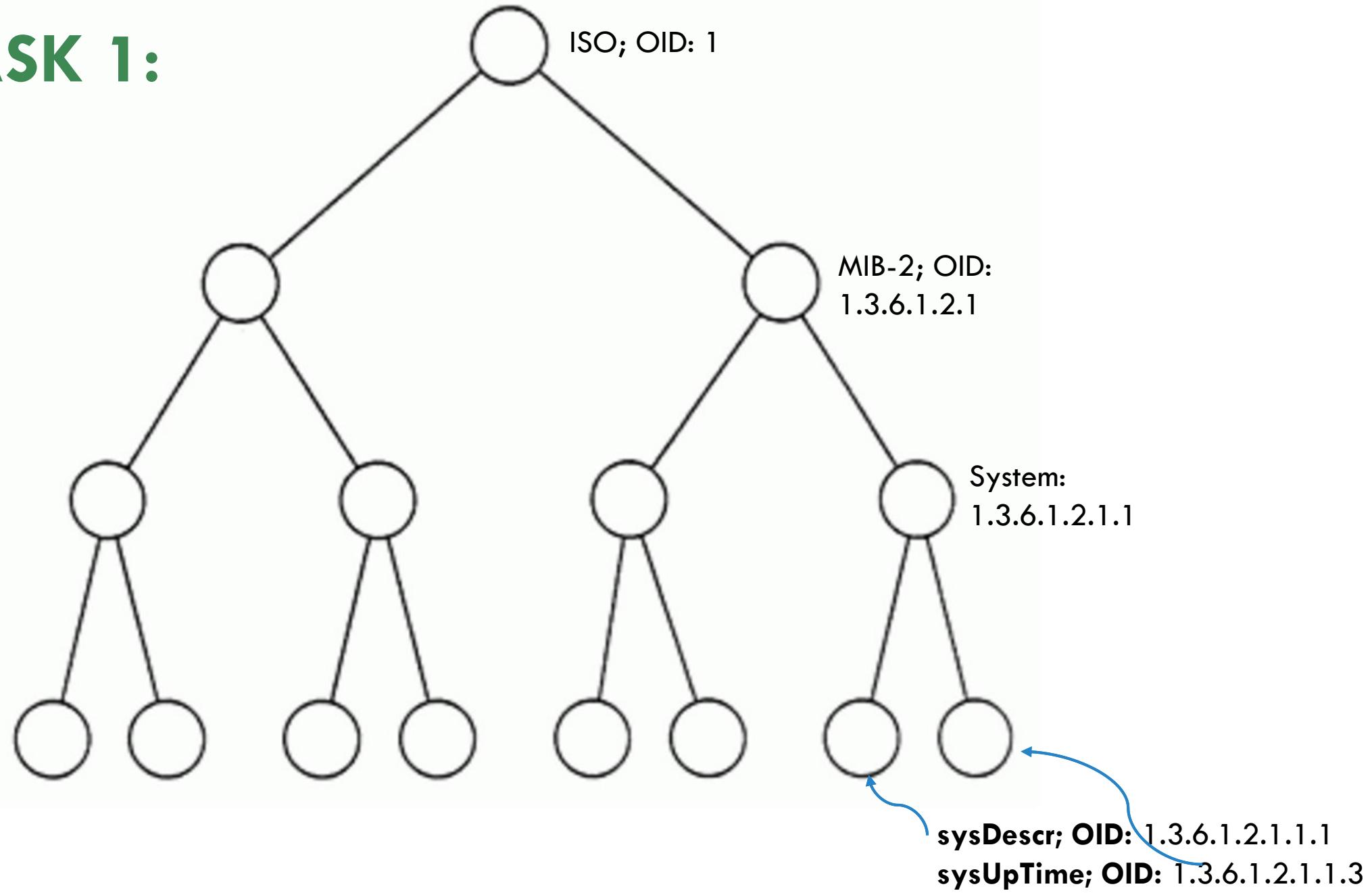
- Kolejność wyszukiwania ma znaczenie.
- Ufamy że elementy pojawiają się w odpowiedniej kolejności. Tj. ufamy że twórcy plików MIB dobrze je spreparowali. Ze importują potrzebne, rodzicielskie elementy wtedy kiedy trzeba.
- ***Propozycja:*** Po napotkaniu elementu; od razu go przetwarzajmy; tj. po napotkaniu obiektu OBJECT IDENTIFIER oraz razu dodawajmy reprezentacje jego do drzewa, z pominięciem tymczasowych struktur danych. (takich jak listy itp.). Przyśpieszy to działanie i zmniejszy zużycie pamięci.
- Dzięki temu unikniemy sytuacji w której np. deklaracje identyfikatora zawiera się w polu DESCRIPTION makra OBJECT-TYPE.

UWAGI DOTYCZĄCE PARSOWANIA

- Kolejność wyszukiwania ma znaczenie. Np. być może najpierw warto sprawdzić importy, a na końcu makra OBJECT-TYPE
- W przypadku makra OBJECT-TYPE, zawiera ono pole DESCRIPTION. Które może zawierać praktycznie dowolne ciągi znaków. Dla tego, rozwiązaniem mogło by być na samym początku wyszukanie całych makr OBJECT-TYPE w strumieniu wejściowym S1, wycięcie ich do jakiegoś bufora B1 zawierającego tylko te struktury. Na pozostałości ciągu S1 (po wycięciu makr) wyszukiwanie innych elementów (takich jak definicje typów danych, importy, deklaracje identyfikatorów)

Dzięki temu unikniemy sytuacji w której np. deklaracje identyfikatora znajdują się w polu DESCRIPTION makra OBJECT-TYPE.

TASK 1:



KLUCZOWE FUNKCJE

- 1. Tworzenie Drzewa
- 2. Wyszukiwanie w drzewie wszerz
 - po nazwie
 - po oid
- 3. Parsowanie i przechowywanie typów danych
- 4. Parsowanie typów z liści
- 5. Zapewnienie powiązania pomiędzy liśćmi a typami
- 6. Atomiczne wartości do wszystkiego co przetwarzamy i uzyskanie dostępu do nich przez gettery/settery
- 7. Ograniczenia typów
- 8. Możliwość dojścia do typu macierzystego
- 9. Parsowanie i przechowywanie informacji i sekwencjach; jedynie w kontekście co w środku
- 10. Importy

OPIS ZADANIA

Przetworzenie wszystkich elementów, umieszczenie w odpowiednich strukturach, umożliwienie atomicznego dostępu do ich właściwości.

Zapewniamy powiązanie pomiędzy typem danych a elementem w drzewie.

W każdym elemencie drzewa, zapisujemy dane go opisujące.

Umożliwiamy atomiczne odpytywanie właściwości obiektów.

TASK1

Potrzebne Narzędzia:

- silnik wyrażeń regularnych np. REGEX
- język programowania np. COBOL
- plik MIB-II – *jako dane wejściowe do testów*

Kompromisy na które pozwalamy:

- brak natywnego, pełnego, wsparcie notacji ASN.1
- to co szczegółowe – „hardkodujemy” – ma działać

REASUMUJĄC, CO PARSER POTRZEBUJE PARSOWAĆ

1) Importy – deklaracje tego co poniżej z innych plików

Proponuje rekurencyjnie przeszukiwać pliki importów, a każdy plik traktować „jednakowo”

2) deklaracje „OBJECT IDENTIFIER” – wprowadza nowe OIDy

3) deklaracje „OBJECT TYPE” – nowe obiekty zarządzalne

4) definicje nowych typów danych

REASUMUJĄC, FUNKCJONALNOŚĆ IMPLEMENTOWANA W 1 ETAPIE

- Obsługa na wejściu dowolnego pliku MIB(testujemy na MIB-2)
- Tworzy drzewo obiektów zarządzalnych, parsując rekurencyjnie importy
- Tworzy listę typów danych
- Umożliwia odnajdywanie obiektów po OID
- Umożliwia odnajdywanie informacji o typie danych znajdującym się pod danym OID

JAK TESTUJEMY? TAK BY DZIAŁAŁO😊

- „Testy jednostkowe” – zapewnią najlepsze pokrycie przypadków
- Interfejs tekstowy/ graficzny – ładne, mniej praktyczne
- W DEBUGGERZE – wybitnie małe pokrycie przypadków, ale robimy to i tak

Etap 2

KODOWANIE BER *CCITT X.209(!)*

Koder / dekoder tworzymy na podstawie tej prezentacji oraz powyższego dokumentu.
Nie na podstawie innych informacji zawartych w Internecie.



ŹRÓDŁO INFORMACJI X-209

Jeśli kiedykolwiek mamy wątpliwości,
otwieramy dokumentacje X.209 i
szukamy sekcji która nas interesuje.

1)

6.5	End-of-contents octets
7	<i>Encoding of a Boolean value</i>
8	<i>Encoding of an integer value</i>
9	<i>Encoding of an enumerated value</i>
10	<i>Encoding of a real value</i>
11	<i>Encoding of a bitstring value</i>
12	<i>Encoding of an octetstring value</i>
13	<i>Encoding of a null value</i>
14	<i>Encoding of a sequence value</i>

13 **Encoding of a null value**

- 2) 13.1 The encoding of a null value shall be primitive.
13.2 The contents octets shall not contain any octets.

Note - The length octet is zero.

Example - If of type NULL, the NULL can be encoded as:

Null	Length
05_{16}	00_{16}

CEL ETAPU 2

Stworzenie klas pozwalających na:

❖ Klasa **CBEREncoder** - sekwencyjne, sterowane, binarne kodowanie danych wejściowych. ORAZ kodowanie pod zadany OID z drzewa MIB. ()

Np.

1. Hej Koder.... „Zakoduj Integer'a o wartości 3” **LUB**

2. Hej Koder... „rozpocznij tworzenie sekwencji”.. Rozpocząłeś? Ok... „teraz zakoduj w środku”:

1. „Wartość liczbową o wartości 1”

2. „Po niej ciąg bajtów o zawartości „ala ma kota”„

3. A po tym... „rozpocznij kolejną sekwencję” a potem....

4. Wszystko zrozumiałeś? Ok.

3. Daj teraz wynikowy ciąg bajtów.

❖ Klasa **CBEREncoder** - **weryfikacja wprowadzonych danych względem obiektu w drzewie Np.** Hej Koder.... „Zakoduj wartość 3243243243243242 dla OID'a 1.2.3.4.5.6”
CBEREncoder : "dzięki klasie CMIBTree (etap 1) utworzonej dzięki CMIBParser (etap 1) sprawdziłem typ danych przypisany do OID'a 1.2.3.4.5.6" – niestety ale podana wartość jest zbyt duża.

CZEMU KODOWANIE BER?

Na urządzeniach może być wykorzystywane kodowanie little-endian, big-endian, bądź też inne.

Dla tego też, wprowadzamy kolejną warstwę abstrakcji, po to aby uniezależnić reprezentacje danych od platformy sprzętowej.

Kodowanie BER jest sprzęgnięte ze składnią ASN.1 (a więc i SMI) poznaną wcześniej.

- 1. Koncepcja widoczności*
- 2. Koncepcja identyfikatora typu danych*
- 3. Koncepcja typu złożonego/prostego*

Powyższe znajdują odniesienie bezpośrednio w metodyce kodowania = po drodze 😊

TYPY KODOWAŃ UŻYWANY W ASN.1

BER (Basic Encoding Rules) – najwięcej możliwości, mamy dostępne różne tryby kodowania

CER (Canonical Encoding Rules) - koduje dane o nieokreślonej długości

DER (Distinguished Encoding Rules) – koduje dane o z góry określonej długości

POJĘCIA

Kodowanie Prymitywne (*Primitive Encoding*) – bajty w polu Zawartość reprezentującą pojedynczą wartość / jedną konkretną zmienną

Kodowanie Złożone (*Constructed Encoding*) – bajty zawarte w polu Zawartość reprezentujące jedną lub kilka niezależnych wartości.

TYPY ZAPISU WARTOŚCI



Typ złożoności określony za pomocą jednego bitu pola Identyfikator

1. Zapis prosty (Primitive), o określonej długości
 - Do kodowania prostych typów danych
2. Zapis złożony (Constructed), o określonej długości danych
 - np. dla SEQUENCE, SEQUENCE [OF], SET
3. Zapis złożony, o nieokreślonej długości
 - np. dla SEQUENCE, SEQUENCE [OF], SET

TYPY DANYCH

- **Typy proste**

To takie które nie składają się z innych składowych

- **Typy złożone**

To takie które zawierają kilka komponentów:
SEQUENCE, SEQUENCE OF, SET, SET OF

OCTET STRING także może być
oznaczony jako złożony w
przypadku kodowania o nieznanej
długości.

Nie obsługiwane w SMI

Przykładowe typy proste z SMI (inne ASN.1 wykreślone):

Type	Tag number (decimal)	Tag number (hexadecimal)
INTEGER	2	02
BIT STRING	3	03
OCTET STRING	4	04
NULL	5	05
OBJECT IDENTIFIER	6	06
SEQUENCE and SEQUENCE OF	16	10
SET and SET OF	17	11
PrintableString	19	13
T61String	20	14
IA5String	22	16
UTCTime	23	17

TYPY PROSTE C.D

Typy proste dzielą się na dwie kategorie:

- Typy łańcuchowe (string)
~~BIT STRING, IA5String, OCTET STRING, PrintableString, T61String, and UTCTime~~
- Typy nie-łańcuchowe
- NULL, INTEGER, Object Identifier

Możemy wysyłać OCTET STRING przy użyciu kodowania o nieznanej
długości, jako typ złożony (Constructed)

Ogólna struktura kodowania BER

Typowe kodowanie Type-Length-Value

Identyfikator/TAG

Długość

Zawartość

Aaaaale...

Identyfikator/TAG

Długość

Zawartość

EOC



W przypadku formy o nieokreślonej długości

Aaaaale.. (kodowanie typu NULL)

Kiedy nie ma co zakodować. Jak np. w przypadku typu NULL

Identyfikator/TAG

Długość

Zawartość



POLE IDENTYFIKATORA

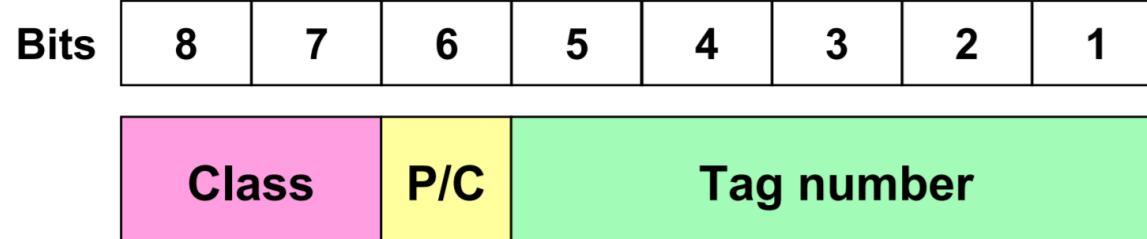
Każdy posiada TAG.

TAG Składa się z :

- identyfikatora klasy
- primitive czy constructed
- nieujemnego identyfikatora

Klasy Tagów:

- uniwersalne – zdefiniowane w X.208: **00b**
- application-specific : **01b**
- prywatne: **10b**



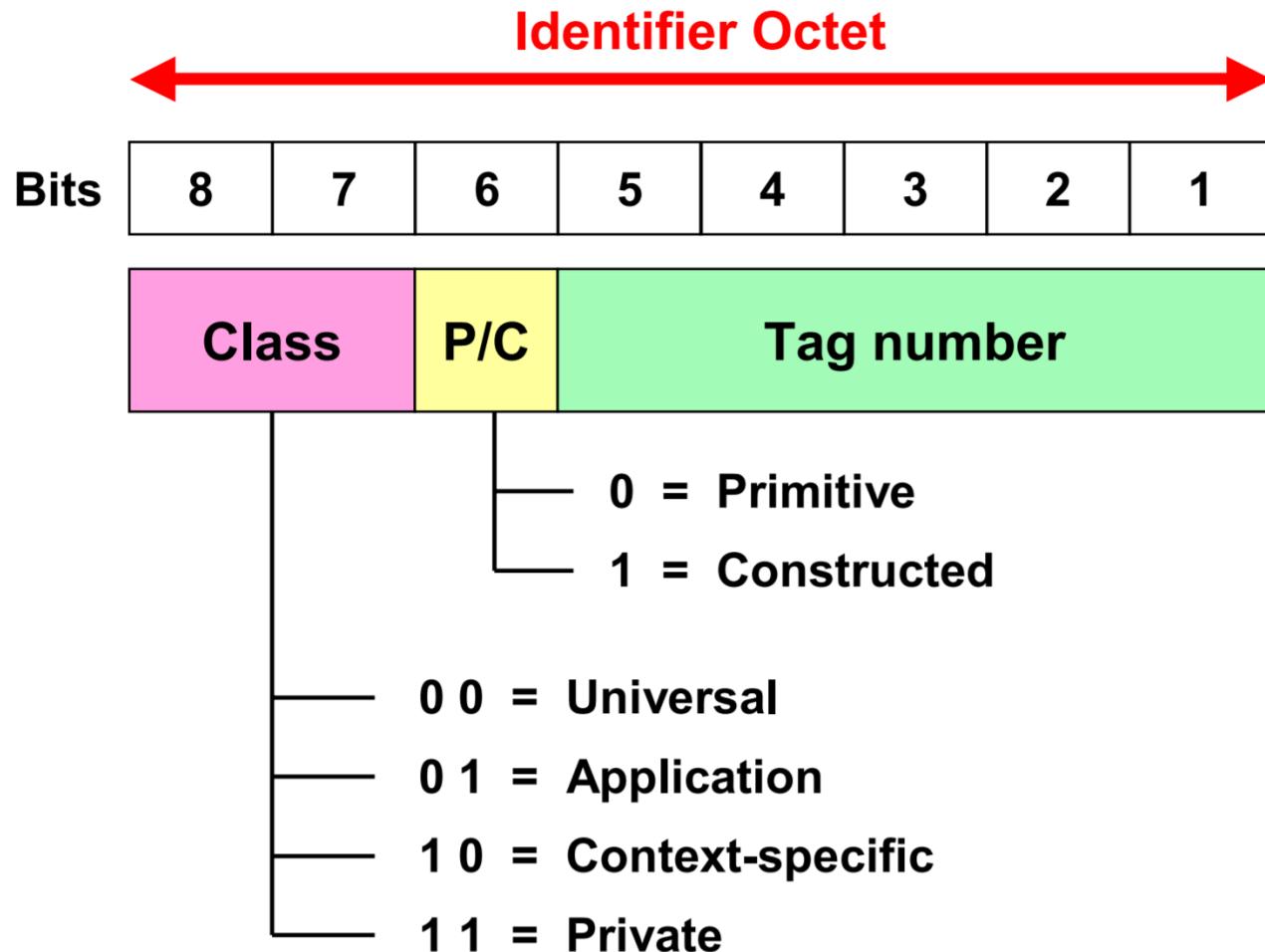
Po co?

Pole Klasy – określa widoczność typu. Tj. w jakim obszarze dany typ istnieje.

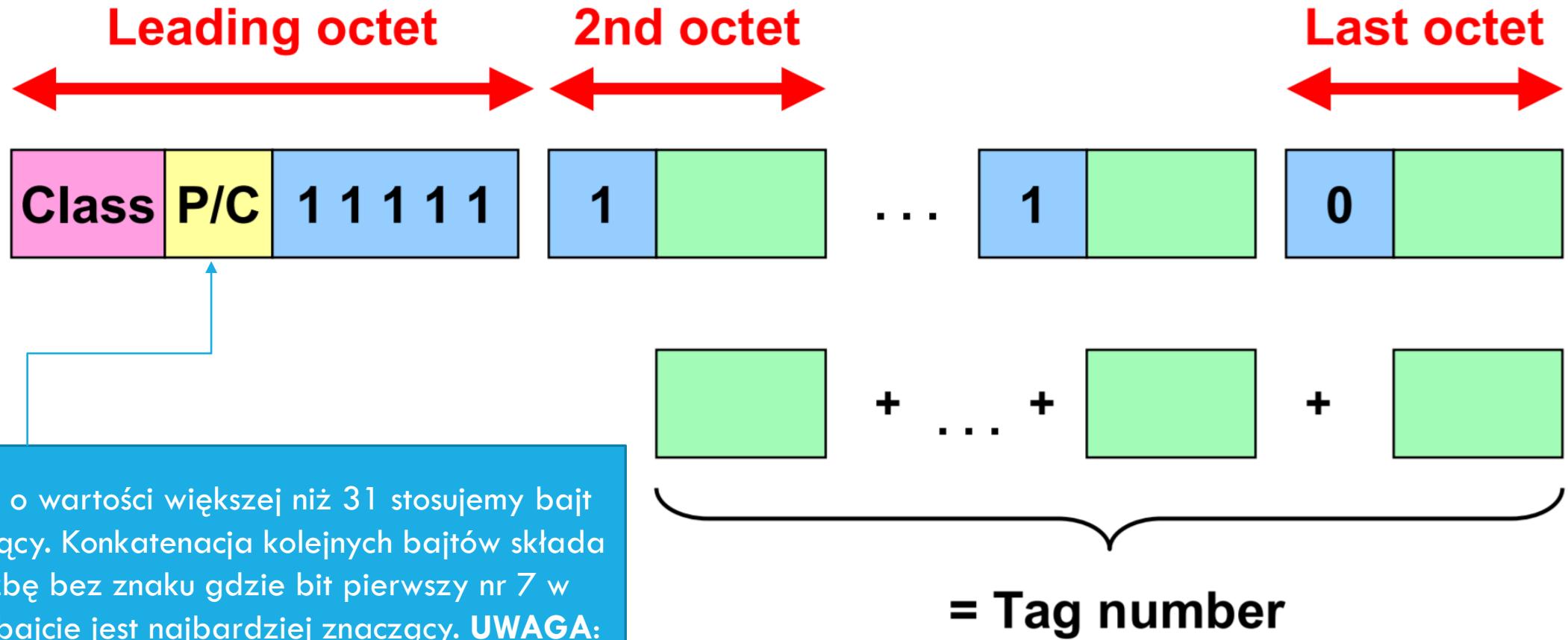
(Patrz [TUTAJ](#) – slajd poprzedniego etapu)

Pole P/C – określa czy kodowany typ jest prosty czy też złożony.

KODOWANIE DLA TAGÓW < 31 (POLE IDENTYFIKATORA)



DLA TAGÓW > 31 (POLE IDENTYFIKATORA)



Dla tag'ów o wartości większej niż 31 stosujemy bajt poprzedzający. Konkatenacja kolejnych bajtów składa się na liczbę bez znaku gdzie bit pierwszy nr 7 w pierwszym bajcie jest najbardziej znaczący. **UWAGA:** pierwszy bit w tych bajtach zarezerwowany jest na oznaczanie ostatniego/lub nie; bajtu.

Pierwszy następujący bajt nie może być cały = 0

KODOWANIE TYPU CHOICE

Typ CHOICE kodujemy używając taga typu danych który rzeczywiście jest przesyłany.

KODOWANIE TYPU BOOLEAN

- Jeśli wartość TRUE => bity pola WARTOŚĆ różne od zera
- Jeśli wartość FALSE => wszystkie bity ustawione na zero

Używamy kodowania prymitywnego.

KODOWANIE INTEGER'A

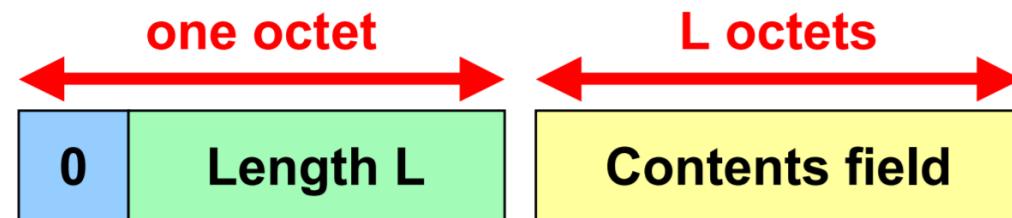
- Używamy kodowania prymitywnego.
- Pole ZAWARTOŚĆ powinno składać się z jednego bądź też wielu bajtów
- Liczby przesyłane są kodowane w systemie U2 (czyli na 99,999999% tak jak na komputerze z którego właśnie korzystasz) – jest to kodowanie łatwe i praktyczne do zapisu liczb ujemnych.

KODOWANIE POLA DŁUGOŚĆ (LENGTH)

- Krótka forma o określonej ilości (< 128 bajtów)
- Długa forma o określonej ilości ($128 < \text{ilość danych} < 2^{1008}$)
 - Nadawca używa reprezentacji o określonej długości jeśli typ danych jest prymitywny
 - Używa określonego lub nie określonego jeśli wszystkie dane do wysłania ma dostępne a typ danych jest złożony.
- Format nieokreślony; koniec danych określony przez EOC
 - Nadawca używa tego typu kodowania jeśli dane nie są od razu dostępne i typ danych jest złożony.

KODOWANIE POLA *LENGTH*

- Krótka forma o określonej ilości (< 128 bajtów)



UWAGA:
K nie może być równe 127

Typy proste lub złożone < 128 bajtów

- Długa forma o określonej ilości ($128 < \text{ilość danych} < 2^{1008}$)



Flagę mamy tutaj ,za darmo' , ,przy okazji' – najstarszy bajt ma ją zawsze ustawioną przy liczbach > 127

Typy proste lub złożone > 128 bajtów

- Format nieokreślony; koniec danych określony przez EOC



Tylko typy złożone

ASN.1 SKŁADNIA PRZYPISANIA WARTOŚCI

Typy Proste:

```
age INTEGER ::= 45  
name UTF8String ::= "bob,,
```

Typu złożone:

Deklaracja sekwencji:

```
MySequence ::= SEQUENCE {  
    age INTEGER,  
    name UTF8String  
}
```

Przypisanie wartości:

```
seq MySequence ::= { 50, "bob" }
```

PRZYKŁAD

Kodowanie Liczb U2:

<http://www.algorytm.edu.pl/systemy-liczbowe/u2.html>

1) Kodujemy wartość dla prostego, podstawowego typu danych **INTEGER**:

yesterday **INTEGER** ::= 127

today **INTEGER** ::= 128

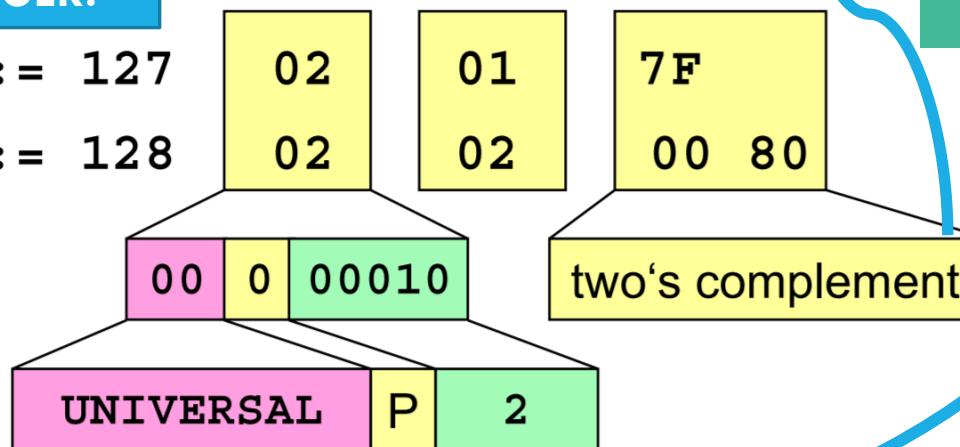
2) Teraz, przy deklaracji typu **DayOfYear** chcemy mieć wpływ na identyfikator typu danych przesyłanej wartości i ustawić go na 17:

DayOfYear ::= [APPLICATION 17] **IMPLICIT INTEGER**

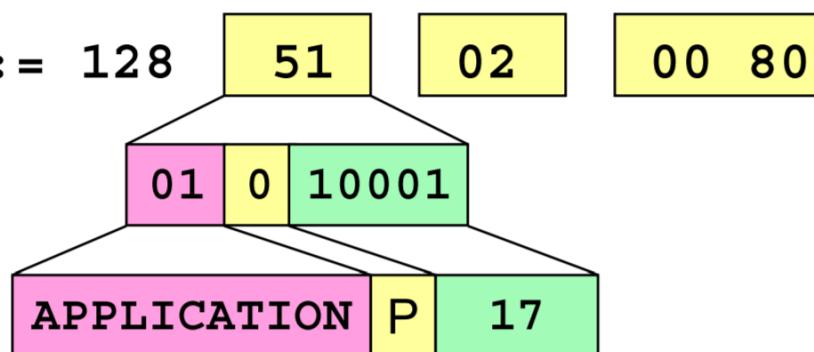
today DayOfYear ::= 128

Typ **DayOfYear** jest używany na kolejnym slajdzie
Podczas tworzenia sekwencji
😊

Type Length Value



Słowo **IMPLICIT** wyjaśni się na kolejnych slajdach. Preczytuje typ kodowania TAG'a.



Ustawienie dedykowanej wartości ID dla Tag'a to tak zwane „otagowanie” typu.

Daje nam to możliwość rozpoznania po stronie odbiorcy danych jakiego typu, specyficznego dla naszej aplikacji, przesyłana wartość dotyczy.

VisibleString - typ danych ASN.1 o ID = 26

Typ danych **DayOfYear** - został zdefiniowany na poprzednim slajdzie

KODOWANIE SEKWENCJI - PRZYKŁAD

Birthday ::= SEQUENCE {
 name VisibleString,
 day DayOfYear
}

Type Definition
UNIVERSAL 16
00 1 10000

myBirthday Birthday ::= {
 name "Jane",
 day 128
}

Value Assignment

Birthday	Length	Contents	BER Encoding	
30	0A			
		VisibleString	Length	Contents
		1A	04	"Jane"
		DayOfYear	Length	Contents
		51	02	00 80

INFO:
Sekwencje SEQUENCE oraz SEQUENCE OF, składają się z kilku innych „trójc” umieszczonych w polu ZAWARTOŚĆ. Pole LENGTH sekwencji, określa całkowitą zsumowaną długość enkapsulowanych typów danych.

UWAGA: Potrzebujemy móc kodować dowolne zagnieżdżenia typów. Np. sekwencja w sekwencji.

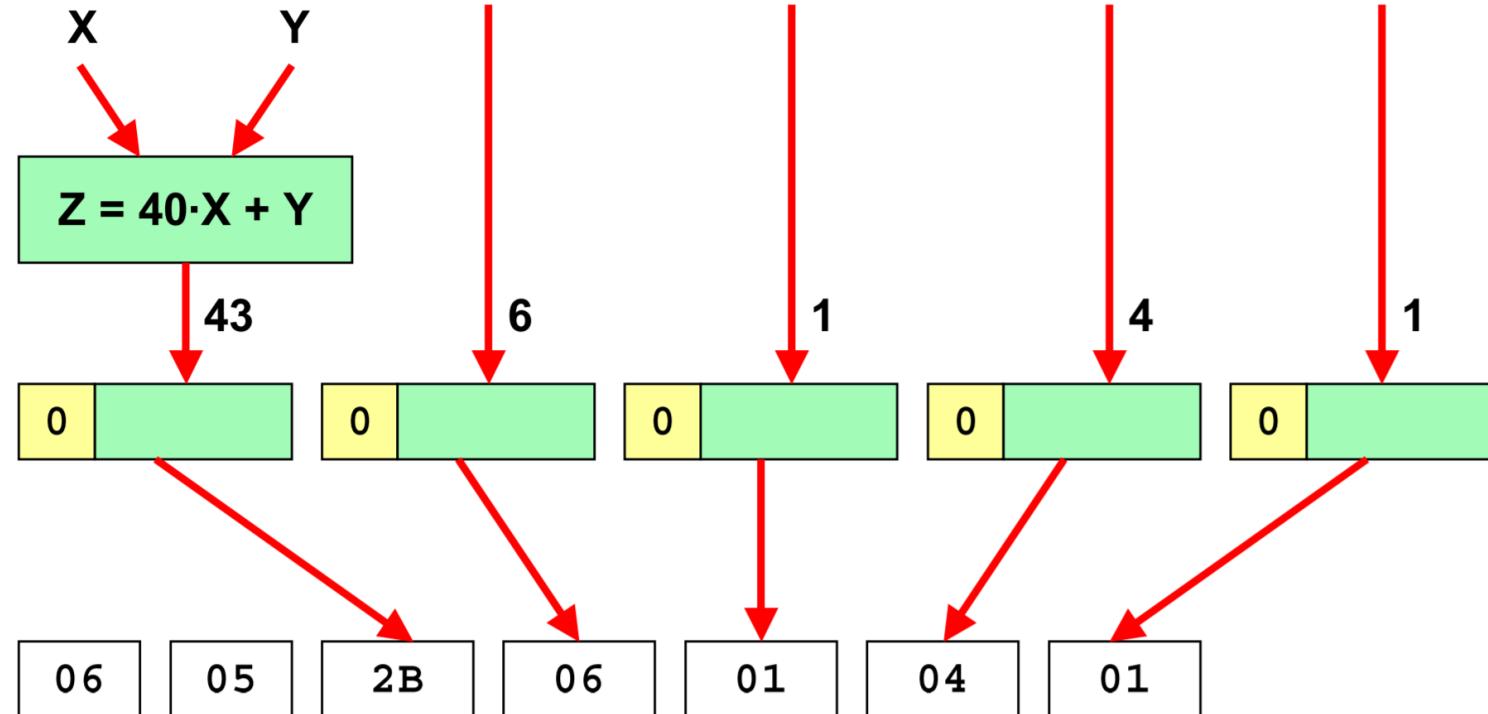
PYTANIA KONTROLNE

- jak zakodujemy LENGTH typu prostego jeśli długość wynosi 12?
- Jak zakodujemy pole LENGTH typu złożonego jeśli długość wynosi 15?
- Jak zakodujemy pole LENGTH typu złożonego jeśli długość wynosi 215?

KODOWANIE OBJECT-IDENTIFIER

enterprise OBJECT IDENTIFIER ::=

{iso(1) org(3) dod(6) internet(1) private(4) 1}



UWAGA: Pierwszy bit określa czy to pierwszy czy ostatni bajt składający się na liczbę. Istotne przy wartościach większych od 2^7 .

Z PLIKU RFC1155, KTÓRE POTRZEBUJEMY OBSŁUGIWAĆ

- `IpAddress ::= [APPLICATION 0] IMPLICIT OCTET STRING (size 4)`
- `Counter ::= [APPLICATION 1] IMPLICIT INTEGER (0..4294967295)`
- `Gauge ::= [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)`

`(..)NetworkAddress, IpAddress, Counter(..)`

PRZYKŁADOWE DANE ZAKODOWANE W BER

- 01, 02, FF 7F (INTEGER, -129)
- 04, 04, 01 02 03 04 (OCTET STRING, – wartość to: 01020304)
- 05 00 (NULL)
- 1A 05 4A 6F T3 65 73 (ciąg znaków “Jones”)
- 30 06, 02 01 03, 02 01 08 (sekwencja dwóch liczb INTEGER)

UNIVERSAL VS IMPLICIT VS EXPLICIT

W składni ASN.1 możemy tworzyć własne typy danych.

Strona odbierająca dane potrzebuje wiedzieć jakiego typu one są, po to aby wiedzieć czego one się dotyczą, jakiego są typu, oraz aby móc sprawdzić czy wartość podana jest z dozwolonego zakresu.

Wysyłając dane zakodowane TAG zawiera informacje o zakodowanym typie danych.

Mamy trzy możliwości:

- Używamy typu uniwersalnego
- Deklarujemy nowy typ danych za pomocą słowa kluczowego IMPLICIT
- Deklarujemy nowy typ danych za pomocą słowa kluczowego EXPLICIT

IMPLICIT VS EXPLICIT

Info: Jeśli zadeklarujemy nowy typ nie podając widoczności, domyślnie przyjęta zostanie widoczność CONTEXT-SPECIFIC.

- **IMPLICIT**

- w przypadku operatora IMPLICIT informacja o macierzystym typie danych jest tracona
- Zajmuje mniej miejsca

- **EXPLICIT**

- W przypadku operatora EXPLICIT zachowujemy informacje o typie macierzystym
- Kodowanie to zajmuje więcej miejsca
- **Będzie wykorzystywane do kodowania sekwencji**

UNIVERSAL VS IMPLICIT VS EXPLICIT

Wartość Liczbowa = 5

A ::= INTEGER

Zakodowane: 02 01 05

B ::= [APPLICATION 4] IMPLICIT INTEGER

Zakodowane: 44 01 05

C ::= [APPLICATION 5] EXPLICIT INTEGER

Zakodowane: 65 03 02 01 05

1) Kodowanie danych oznaczonych jako Explicit tworzy się kapsułkując reprezentację potomka w polu DANE typu macierzystego. Tworząc tym samym typ złożony (constructed).

W przypadku kodowania złożonego (constructed encoding)
Bit Constructed (6ty w polu TAG ustawiony jest na 1).

UNIVERSAL VS IMPLICIT VS EXPLICIT C.D

B ::= [4] IMPLICIT INTEGER

Zakodowane: 84 01 05

C ::= [5] EXPLICIT INTEGER

Zakodowane : A5 03 02 01 05

1) Jeśli nie podamy widoczności typu (dla IMPLICIT lub EXPLICIT)
– domyślną kasą będzie **Context-Specific**

2) Jeśli typ jest ‘otagowany’ (zawiera np. [5]) wartość nie ma ani implicit ani explicit
- przyjmujemy kodowanie explicit i ustawiamy bit **constructed**

3) Jeśli zadeklarujemy nowy typ nie podając widoczności
- domyślnie przyjęta zostanie widoczność **CONTEXT-SPECIFIC**.

4) Nowy typ danych jeśli nie posiada własnego identyfikatora czy też widoczności (alias); typ ten dziedziczy je po typie macierzystym.
A więc, gdy kodujemy nowy, ‘wydziedziczony’ typ w trybie explicit, będzie on posiadał ID typu macierzystego.

RULES

1) Jeśli nie podamy widoczności typu (dla IMPLICIT lub EXPLICIT) – domyślną kasą będzie Context-Specific

2) Jeśli typ jest ‘otagowany’ (zawiera np. [2]) oraz nie ma ani implicit'a ani explicit'a - przyjmujemy kodowanie explicit i ustawiamy bit constructed

3) Jeśli zadeklarujemy nowy typ nie podając widoczności - domyślnie przyjęta zostanie widoczność CONTEXT-SPECIFIC.

Type1 ::= VisibleString

VisibleString to typ danych ASN.1 o ID 26.

VisibleString
1A₁₆

Length
05₁₆

Contents
4A6F6E6573₁₆

Type2 ::= [APPLICATION 3] IMPLICIT Type1

[Application 3]
43₁₆

Length
05₁₆

Contents
4A6F6E6573₁₆

Type3 ::= [2] Type2

UWAGA:
kodowanie EXPLICIT

[2]
A2₁₆
[Application 3]
43₁₆

Length
07₁₆
Length
05₁₆

Contents
4A6F6E6573₁₆

Type4 ::= [APPLICATION 7] IMPLICIT Type3

[Application 7]
67₁₆
[Application 31]
43₁₆

Length
07₁₆
Length
05₁₆

Contents
4A6F6E6573₁₆

Type5 ::= [2] IMPLICIT Type2

[2]
82₁₆

Length
05₁₆

Contents
4A6F6E6573₁₆

DO TESTOWANIA

<http://asn1-playground.oss.com/>

Uwaga: skrypt na powyższej stronie nie koduje prawidłowo wartości wewnętrz sekwencji. Wartości wewnętrz sekwencji powinny zawierać taki wynikające z typu danych tak jednak nie jest (TAGi zastępowane są indeksami)

PRZYKŁADOWO

Schema: Enter manually ▾

```
World-Schema DEFINITIONS AUTOMATIC TAGS ::=  
BEGIN  
C ::= [5]EXPLICIT INTEGER  
  
END
```

Data: HEX text ▾ T

Compile

Decode ▶▶▶

DATA: ENCODE

Enter a Value (in the ASN.1 Value Notation format) for one of the Types defined in the Schema. Click Encode. Various encoded formats will be available as links for downloading.

Value: JSON ▾ Type: C ▾

5

CONSOLE OUT

Options:

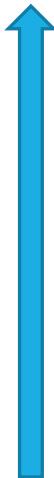
Strict synt.

```
ASN1STEP: Encoding of  
Encoding to the file '  
tag = [5] constructed;  
C INTEGER: tag = [UN  
5  
Encoded successfully i  
A5030201 05
```

Dobrze zakodowaliśmy

TASK 2: Wymagania funkcjonalne

ISO; OID: 1



2) Klasa kodera daje możliwość zakodowania wartości atomowych jak i złożonych (np. sekwencji zawierającej kolejne sekwencje sekwencji.)

3) Przed rozpoczęciem kodowania wartości atomowej program daje możliwość podania OID.

4) **Wartość OID może ale nie musi zostać podana.** Przy podaniu OID sprawdzana jest poprawność podawanych danych. Program potrafi zweryfikować poprawność podanej wartości, ORAZ wygenerować kodowanie BER dla danej instancji typu.

Info: Stanowi to swego rodzaju powiązanie pomiędzy funkcjonalnością kodera a validatora.

UWAGA: Maksymalnie można dostać trzy słoneczka.

1) Potrzebujemy móc kodować dowolne zagnieżdżenia typów.
Np. sekwencja w sekwencji.

Dla tego powinna też być możliwość rozpoczęcia sekwencji, podania danych wewnętrz, oraz zakończenia sekwencji.

1.3.6.1.2.1

Alternatywą dla takiego interaktywnego testowania jest Sztuczne implementowanie testów jednostkowych.

1.3.6.1.2.1.1
Stwórzmy „validatora” w osobnej klasie? Przyda się nam później. Kiedy?

sysDescr; OID: 1.3.6.1.2.1.1.1
sysUpTime; OID: 1.3.6.1.2.1.1.3

UWAGA: Etap ten jest niezależny funkcjonalnie od etapu poprzedniego. Tworzymy uniwersalny koder BER. Koder BER nie wie **nic** o protokole SNMP czy drzewie MIB. Powinien kodować dowolne elementy BER.

KLUCZOWE FUNKCJE

❖ **Klasa CBERCoder – (przykład:)** podstawową funkcją niech będzie funkcja kodująca biorąca na wejście: ID tag'a, bool oznaczający czy to constructed i zawartość jako wektor bajtów np. encode(ID,constructed,array). Pozostałe funkcje wewnętrzne wyższego rzędu z niej korzystają. I tak np. możemy mieć funkcje publiczne encodeInteger(..,X), encodeByteArray(..,X) które konwertują X na wektor bajtów i wywołują encode(). Do tego funkcje publiczne startSequence(ID) uwaga na ID, endSequence()

- Kodowanie wartości prostych
- Kodowanie sekwencji (i dowolnych zagnieżdżeń wartości w środki w tym innych sekwencji)
- Kodowanie w trybie ze wspomaganiem klasy **CDataVerifier** z podaniem OID i wartości

❖ **Klasa CDataVerifier:**

- Sprawdza czy podana wartość zgodna z typem w liściu o zadanym OID

PUNKTY KONTROLE



- Kodowanie Integer'a, OCTET_STRING'a , bool'a , null'a
- Kodowanie gdy ID Tag'a (typu zmiennej) większe niż 31
- Kodowanie gdy zawartość pola Content większe niż 128 bajtów
- Kodowanie OID gdy ktoreś ID większe niż 2^7 (np. 1.2.123123.1)
- Kodowanie IMPLICIT
- Kodowanie EXPLICIT
- **Iteracyjne** kodowanie sekwencji, zagnieżdzonych sekwencji (tj. rozpocznij, umieść integer'a, umieść OCTET_STRING'a, rozpocznij kolejną sekwencję, daj ciąg wynikowy)
- Kodowanie wybranego elementu z drzewa MIB przy weryfikacji wprowadzonych danych (czy zgodne z zakresem typu elementu w drzewie)

TASK 2:



Przy testowaniu sprawdzaniu podawanych kodowanych wartości, potrzebujemy odczytać:

Typ danych – Integer
do sprawdzenia czy wartość podana jest prawidłowa

Sprawdzamy sposób kodowania oraz zapisany TAG:

typ: Implicit
wartość TAG'u: 3, wykorzystamy podczas kodowania

- Validator (jeśli podany OID) : Odczytuje ograniczenia (0...429467295)

Wykorzystane do sprawdzenia czy wartość która chcemy wygenerować jest poprawna, podejmuje decyzję czy zakodujemy czy nie

- Wartość zostaje zakodowana

ISO; OID: 1

MIB-2; OID:
1.3.6.1.2.1

System:
1.3.6.1.2.1.1

sysDescr; OID: 1.3.6.1.2.1.1.1
sysUpTime; OID: 1.3.6.1.2.1.1.3

UWAGA: Potrzebujemy móc kodować dowolne zagnieżdżenia typów. Np. sekwencja w sekwencji.

ETAP 3

DEKODER BER



CEL ETAPU 3

TO: Stworzenie klasy pozwalającej na „dobranie się” do elementów już zakodowanych.

→ **Wejście:** Ciąg bajtów – dane zakodowane w BER

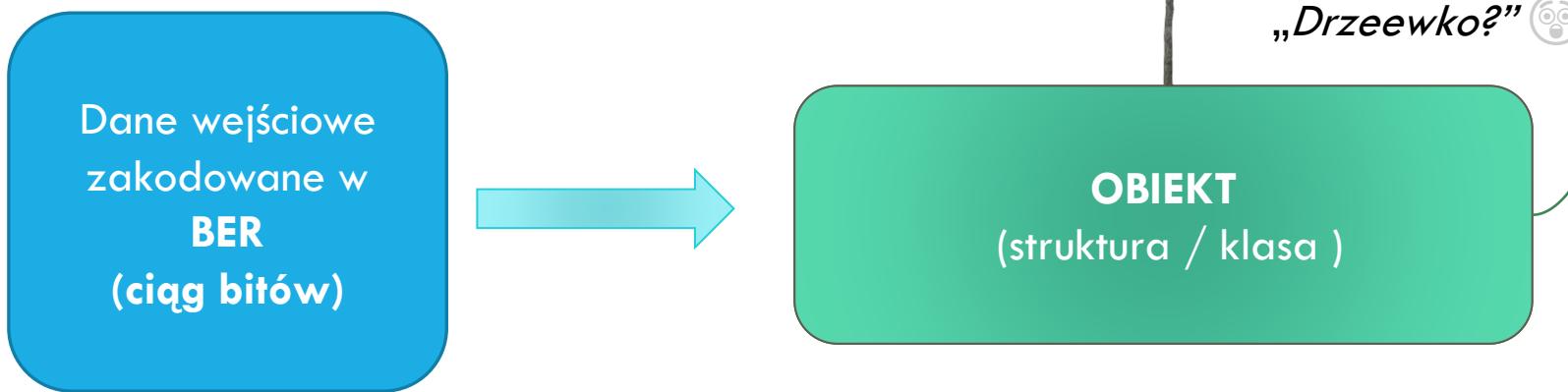
Wyjście: → Drzewko() , zawierające obiekty reprezentujące każdą zakodowaną wartość. (Hierarchia )

Kolokwialne zobrazowanie:

Użytkownika(programista ): „Hej Koder.... mam tu ciąg bajtów; chce wiedzieć co w środku. Rozpakuj mi to w ładne zrozumiałe dla mnie drzewko gdzie każdy ‚obiekt’ zawiera wszystkie właściwości zakodowanego elementu”

Koder  : "O, jak miło że prosiś(ಠಠ) zabieram się do roboty! gdyby coś było w danych wejściowych nie tak - dam znać; będzie ok- przedstawię drzewko"

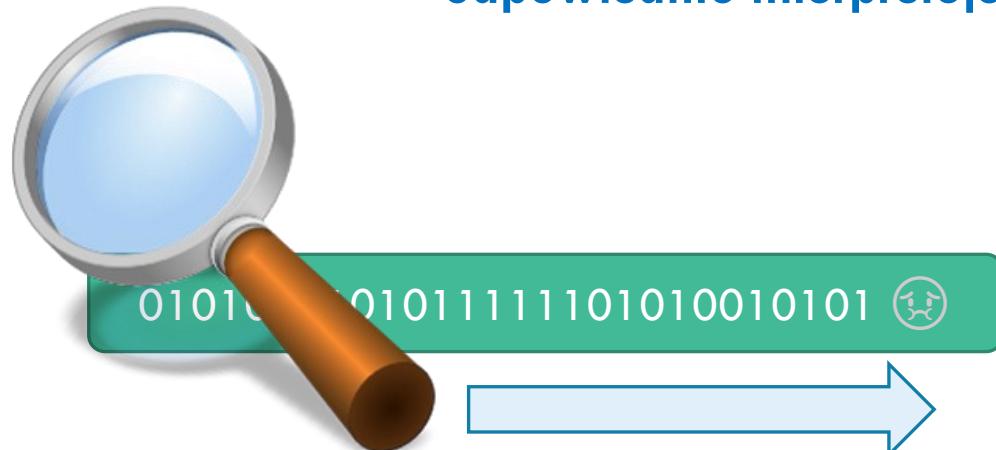
TASK 3 - DEKODER BER



Propozycja: Dane dekodujemy do postaci **drzewa** składającego się z jednego (typy prymitywne) lub też wielu (sekwencje) elementów.

TASK 3 - DEKODER BER

1. Dane wejściowe czytamy od lewej do prawej, odpowiednio interpretujemy poszczególne pola (bajty).



2. Wykorzystujemy wiedzę (💡), że każdy obiekt składa się z trzech pól:
TAG, DŁUGOŚĆ, WARTOŚĆ, z których każde może składać się z wielu bajtów.

3. Odpowiednio interpretujemy pole **TAG** (dla wartości > 31 zapis na wielu bajtach ☺)

4. Odpowiednio interpretujemy pole **DŁUGOŚĆ** (długa lub krótka forma zapisu)

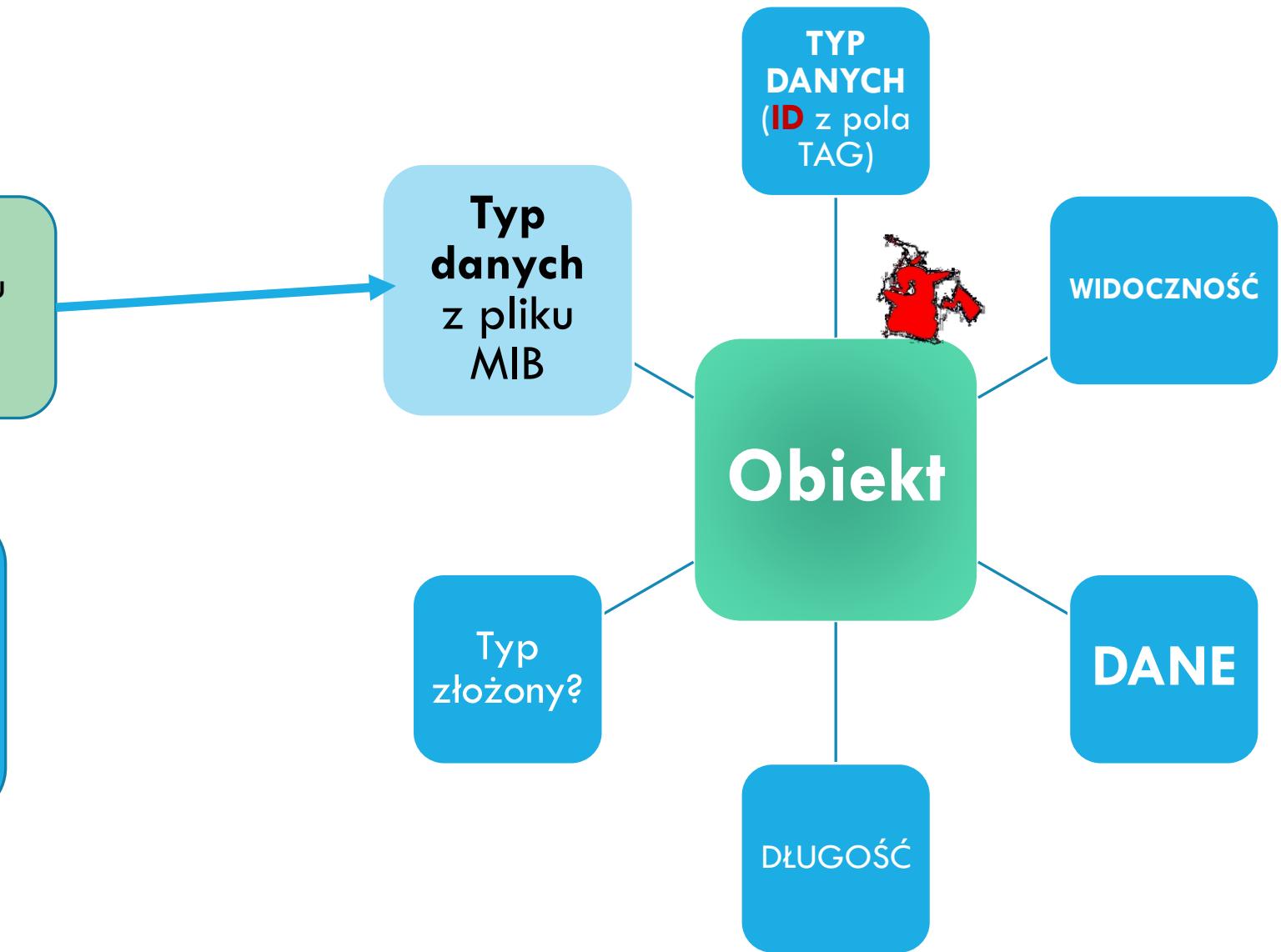
5. Bajty pola **WARTOŚĆ** sklejamy od lewej do prawej. Pierwszy bit z lewej będzie najstarszy. Następnie interpretujemy dane zgodnie z typem.



WYDOBYWAMY ZAWARTE W OBIEKTACH WŁAŚCIWOŚCI

Propozycja: Aby od razu tutaj dokonać próby odszukania typu danych po **ID** na naszej liście rozpoznanych typów z **Etapu 1**

Propozycja: Każdą właściwość przechowujemy jako osobny element struktury / obiektu.



ODPOWIEDNIA INTERPRETACJA DANYCH

- **Propozycja:** Umożliwiamy odczyt pola DANE w odpowiedniku dla wybranego języka wyższego poziomu (mowa o polu zawierającym „zawartość”)

Obiekt 1
(element drzewa)

Co zawierasz?



INFO: Domyślnie wszystko możemy przetrzymywać w polu DANE jako tablica bajtów. Potrzebujemy natomiast móc zinterpretować wartość. Sprawdzić czy jest w odpowiednim zakresie wg. Wymagań typu danych z MIB'a.

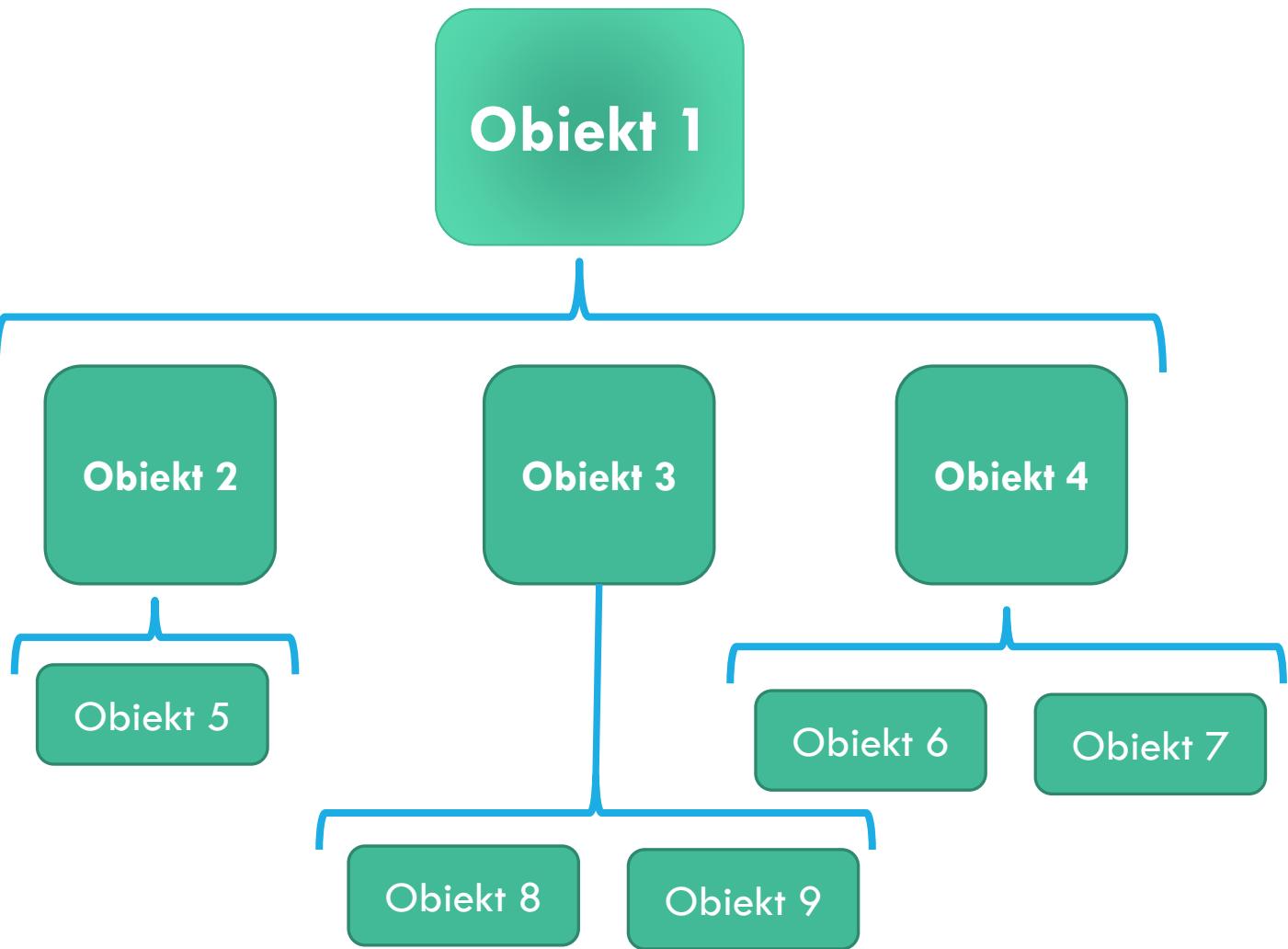
OBIEKT ELEMENTEM DRZĘWA

Dane dekodujemy jako drzewo składające się z jednego (typy prymitywne) lub też wielu (sekwencje) elementów.

Dane wejściowe zakodowane w BER



Każdy obiekt zawiera wskaźnik na kolejny obiekt w nim się znajdujący lub też NULL (jeśli to obiekt prosty)



CO NA WEJŚCIU CO NA WYJŚCIU?

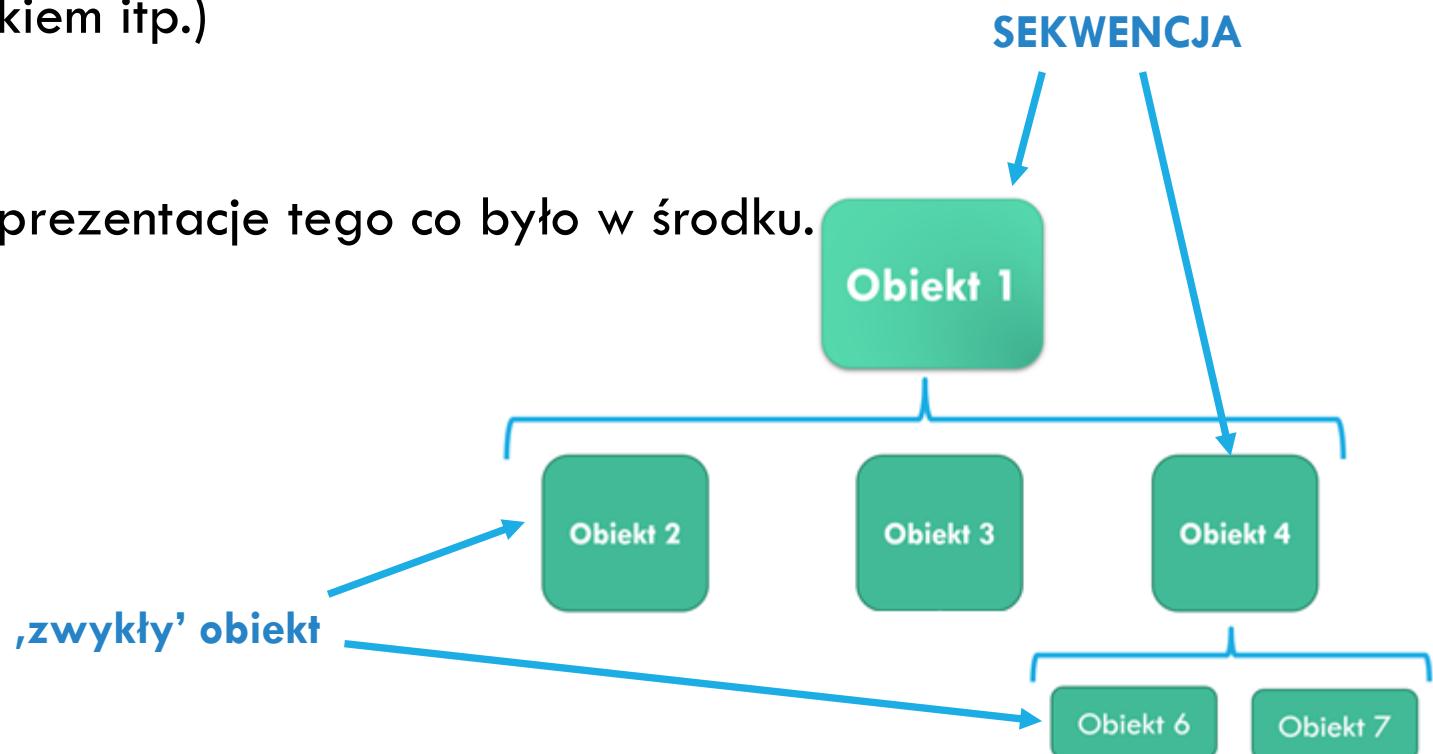
WEJŚCIE:

- klasa obsługuje na wejściu dowolne dane zakodowane w BER

(typy proste, sekwencje, sekwencje sekwencji dowolne kombinacje, jeśli dane nieprawidłowo, informuje, rzuca wyjątkiem itp.)

WYJŚCIE:

- struktura drzewiasta zawierająca reprezentacje tego co było w środku.



PUNKTY KONTROLNE



- ❖ Dekodowanie integer'a
- ❖ Dekodowanie octet-string'a
- ❖ Dekodowanie obiektu w trybie implicit/explicit (informacja o ID typu macierzystego)
- ❖ Dekodowanie sekwencji
- ❖ Dekodowanie sekwencji zawierającej w środku inne wartości (w tym sekwencje)
- ❖ Próba powiązania zdekodowanego elementu z typami danych z **Etapu 1** (jedyny punkt styczny z poprzednimi etapami)

ETAP 4

PDU

(PROTOCOL DATA UNIT)

FORMAT PAKIETU DANYCH

CEL

Celem jest:

- ❖ stworzenie funkcjonalności pozwalającej na dekodowanie wektora bajtów do postaci klasy/struktury reprezentującej PDU protokołu SNMP. Pomysł: funkcja statyczna:

```
static std::shared_ptr<CPDU> CPDU::instantiate(std::vector<uint8_t> packedBytes)
```

- ❖ Stworzenie funkcjonalności pozwalającej na zakodowanie klasy/struktury reprezentującej PDU protokołu SNMP do postaci wektora bajtów. Pomysł:

```
std::vector<uint8_t> CPDU::getPackedData()
```

- ❖ Stworzenie funkcjonalności/klasy przetwarzającej PDU - Wykorzystanie (opcjonalnie wcześniej stworzonego) validatora danych do sprawdzenia czy dane otrzymywane są w zakresie. Na ostatnim etapie będziemy uposażać ten element o interakcje z systemem operacyjnym.

Inaczej mówiąc serializacja/deserializacja obiektów przy pomocy kodera/dekodera który wcześniej sami stworzyliśmy



PAKIET – DEFINICJA W RFC 1157

```
Message ::=  
SEQUENCE {  
    version          -- version-1 for this RFC  
    INTEGER {  
        version-1(0)  
    },  
  
    community       -- community name  
    OCTET STRING,  
  
    data             -- e.g., PDUs if trivial  
    ANY              -- authentication is being used  
}
```

```
PDUs ::=  
CHOICE {  
    get-request      GetRequest-PDU,  
  
    get-next-request GetNextRequest-PDU,  
  
    get-response     GetResponse-PDU,  
  
    set-request      SetRequest-PDU,  
  
    trap             Trap-PDU  
}
```

Protokół zdefiniowany jest za pomocą ASN.1. Widzimy że różne typy pakietów są zdefiniowane za pomocą CHOICE. Za jednym razem możemy wybrać jeden PDU z kilku dostępnych.

FORMATY PDU/TRANSMISJA RFC 1157 [1]

Identyfikator odróżniający typy pakietów

```
GetRequest-PDU ::=  
[0]  IMPLICIT SEQUENCE {  
    request-id  
        RequestID,  
  
    error-status          -- always 0  
        ErrorStatus,  
  
    error-index           -- always 0  
        ErrorIndex,  
  
    variable-bindings  
        VarBindList  
}
```

```
GetResponse-PDU ::=  
[2]  IMPLICIT SEQUENCE {  
    request-id  
        RequestID,  
  
    error-status          ErrorStatus,  
  
    error-index           ErrorIndex,  
  
    variable-bindings  
        VarBindList  
}
```

FORMATY PDU/TRANSMISJA RFC 1157 [2]

Identyfikator odróżniający typy pakietów

SetRequest-PDU ::=
[3]

```
IMPLICIT SEQUENCE {
    request-id
        RequestID,
    error-status          -- always 0
        ErrorStatus,
    error-index           -- always 0
        ErrorIndex,
    variable-bindings
        VarBindList
}
```

GetNextRequest-PDU ::=
[1]

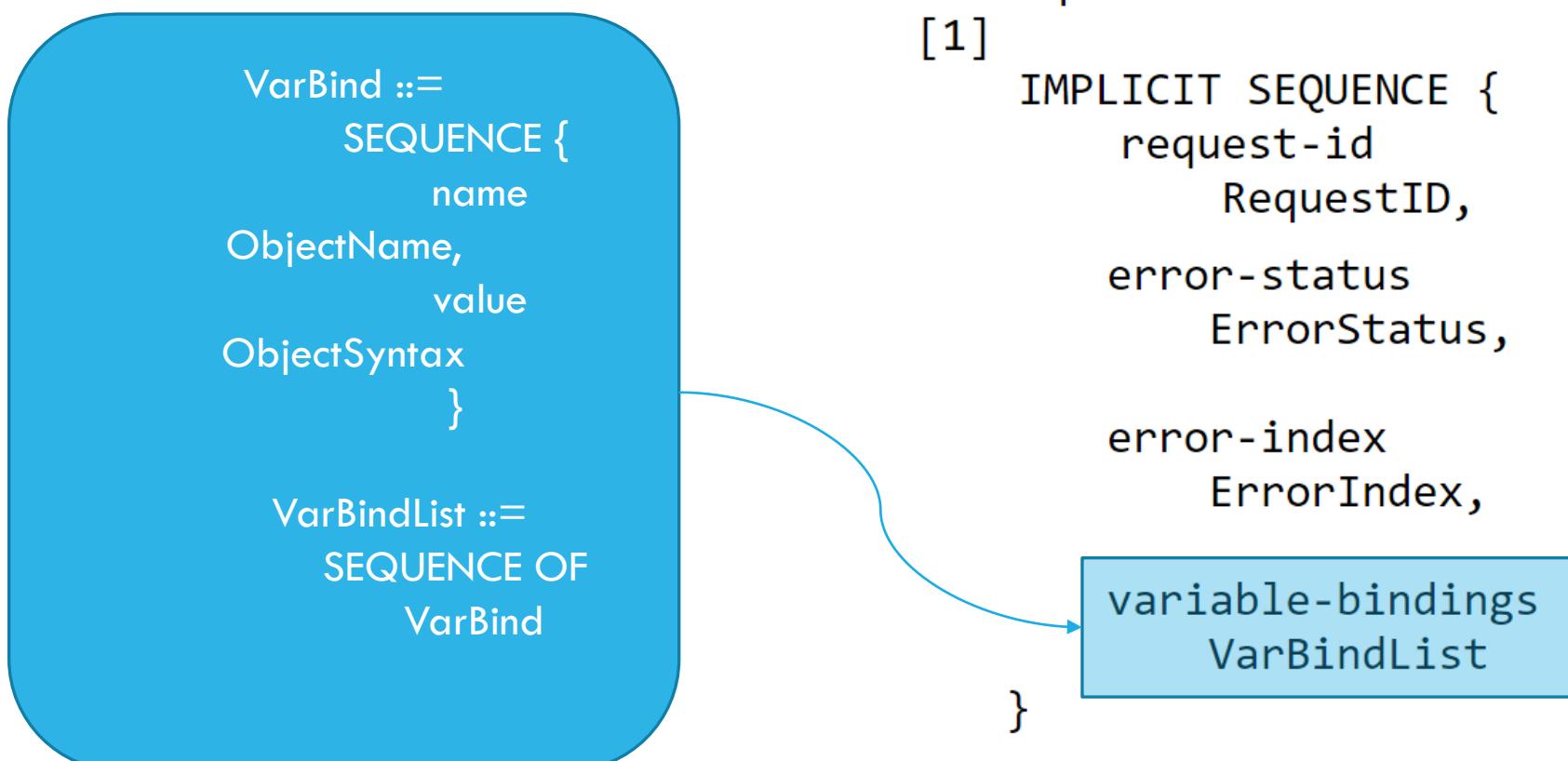
```
IMPLICIT SEQUENCE {
    request-id
        RequestID,
    error-status          -- always 0
        ErrorStatus,
    error-index           -- always 0
        ErrorIndex,
    variable-bindings
        VarBindList
}
```

GETRESPONSE-PDU RFC 1157 [1]

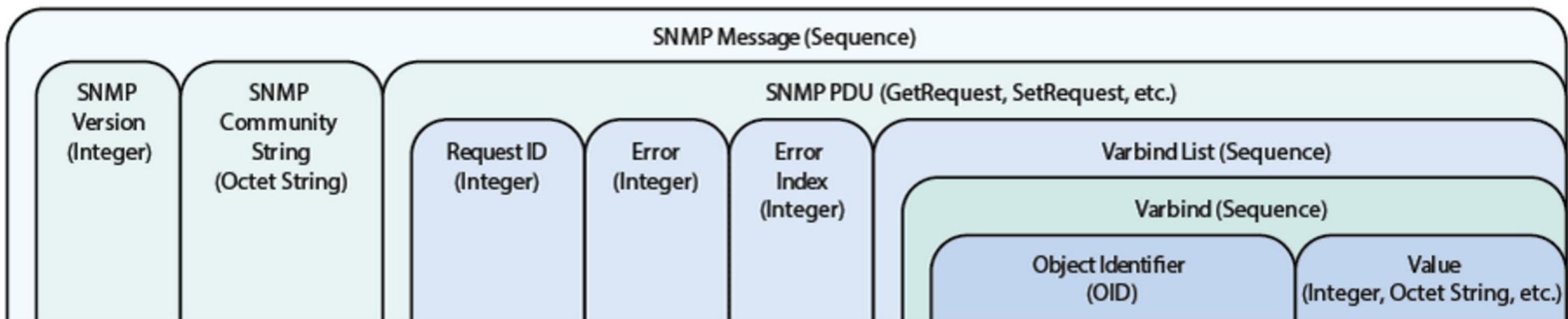
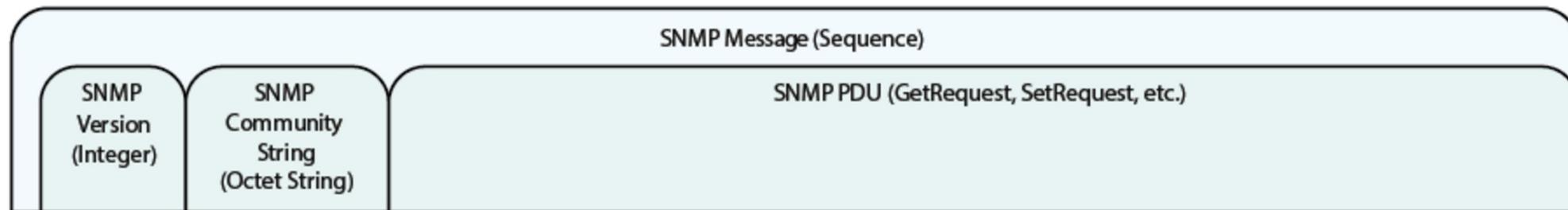
```
GetResponse-PDU ::= [2]
  IMPLICIT SEQUENCE {
    request-id
      RequestID,           ← Idenfikator ZAPYTANIA na które odpowiadamy
    error-status
      ErrorStatus,          ← Idenfikator błędu (jeśli wystąpił)
    error-index
      ErrorIndex,           ← Indeks elementu na liście VarBindList który spowodował błąd
    variable-bindings
      VarBindList           ← Lista par <OID , WARTOŚĆ >, której zwracamy do klienta
  }
```

GetResponse-PDU ::= [2]
IMPLICIT SEQUENCE {
 request-id
 RequestID, ← Idenfikator ZAPYTANIA na które odpowiadamy
 error-status
 ErrorStatus, ← Idenfikator błędu (jeśli wystąpił)
 error-index
 ErrorIndex, ← Indeks elementu na liście VarBindList który spowodował błąd
 variable-bindings
 VarBindList ← Lista par <OID , WARTOŚĆ >, której zwracamy do klienta
}

FORMATY PDU/TRANSMISJA RFC 1157 [2]



STRUKTURA PAKIETU *SNMP*



STRUKTURA PAKIETU *SNMP*

SNMP Message Type = Sequence, Length = 44											
SNMP Version			SNMP Community String			SNMP PDU					
Type = Integer	Type = Octet String	Type = Object Identifier	Type = Integer	Type = Integer	Type = Integer	GetRequest, Length = 30					
Length = 1	Length = 7	Length = 16	Length = 1	Length = 1	Length = 1	Varbind List, Length = 19					
Value = 0	Value = private	Value = 1.3.6.1.4.1.2680.1.2.7.3.2.0	Request ID	Error	Error Index	Varbind, Length = 17					
30 2C	02 01 00	04 07 70 72 69 76 61 74 65	A0 1E	02 01 01	02 01 00	02 01 00	30 13	30 11	06 0D 2B	06 01 04 01 94 78 01 02 07 03 02 00	05 00

SNMP message	Type	0x30	Sequence
	Length	0x2B	Length: 43
	Version	0x02 0x01 0x00	Integer Length: 1 Value: 0
	Community	0x04 0x06 0x70 0x75 0x62 0x6C 0x69 0x63	Octet String Length: 6 Value: public
Data PDU	SNMPv1 PDU type	0xA0	GetRequest PDU
	PDU length	0x1E	Length: 30
	Request ID	0x02 0x04 0x37 0x9C 0x57 0x89	Integer Length: 4 Value:
	Error Status	0x02 0x01 0x00	Integer Length: 1 Value: 0
	Error Index	0x02 0x01 0x00	Integer Length: 1 Value: 0
	VarBind List	0x30	Sequence
	Length	0x10	Length: 16
	VarBind 1	0x30	Sequence
	Len 1	0x0E	Length: 14
	OID 1	0x06 0x0A 0x2B 0x06 0x01 0x04 0x01 0x82 0x99 0x5D 0x02	Object identifier Length: 10 Value:
	Value 1	0x05 0x00	NULL Length: 0

IMPLEMENTACJA FUNKCJI GENERUJĄCEJ PAKIET

- GetRequest
- GetResponse
- SetRequest
- GetNextRequest

IMPLEMENTACJA FUNKCJI GENERUJĄCEJ PAKIET

Propozycja: Funkcja powinna przyjmować identyfikator typu pakietu:

- GetRequest
- GetResponse
- SetRequest
- GetNextRequest

ORAZ, listę par *< OID, Wartość >*

ORAZ, requestID

ORAZ, ErrorStatus

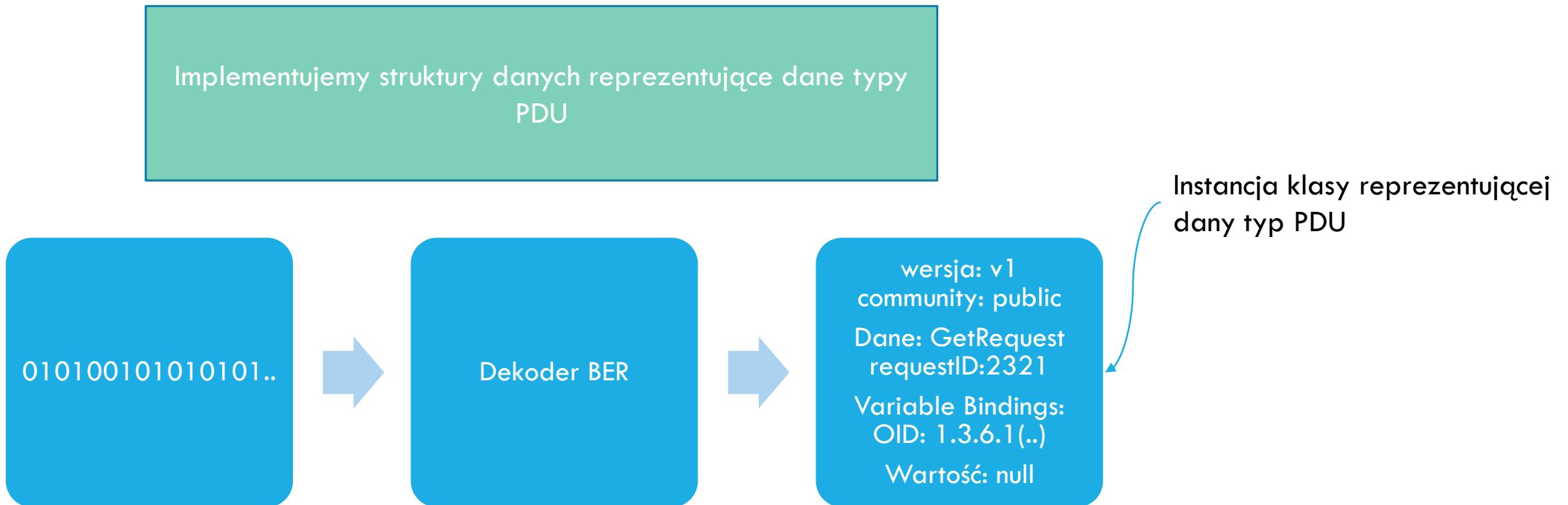
ORAZ, ErrorIndex

GetResponse-PDU ::=
[2]
IMPLICIT SEQUENCE {
request-id
RequestID,
error-status
ErrorStatus,
error-index
ErrorIndex,
variable-bindings
VarBindList
}

KLASY PAKIETÓW



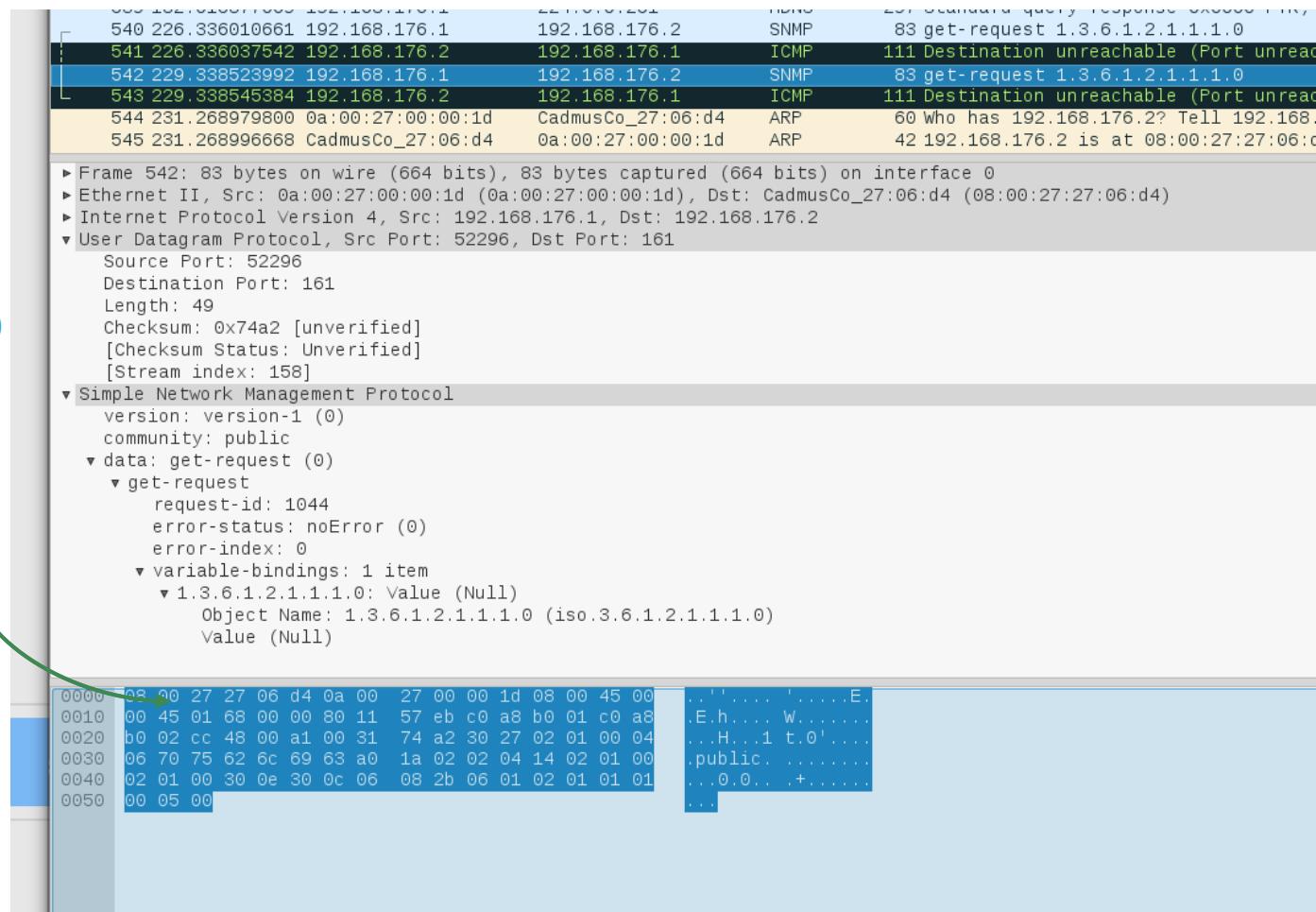
KLASY PDU



SKĄD BIERZEMY DANE WEJŚCIOWE DO TESTÓW?

- Nasłuchujemy na lokalnym interfejsie Loopback za pomocą programu Wireshark
- Wykonujemy zapytanie SNMPGET do lokalnego komputera np:
`snmpget -mALL -v1 -cpublic 127.0.0.1 sysName.0`

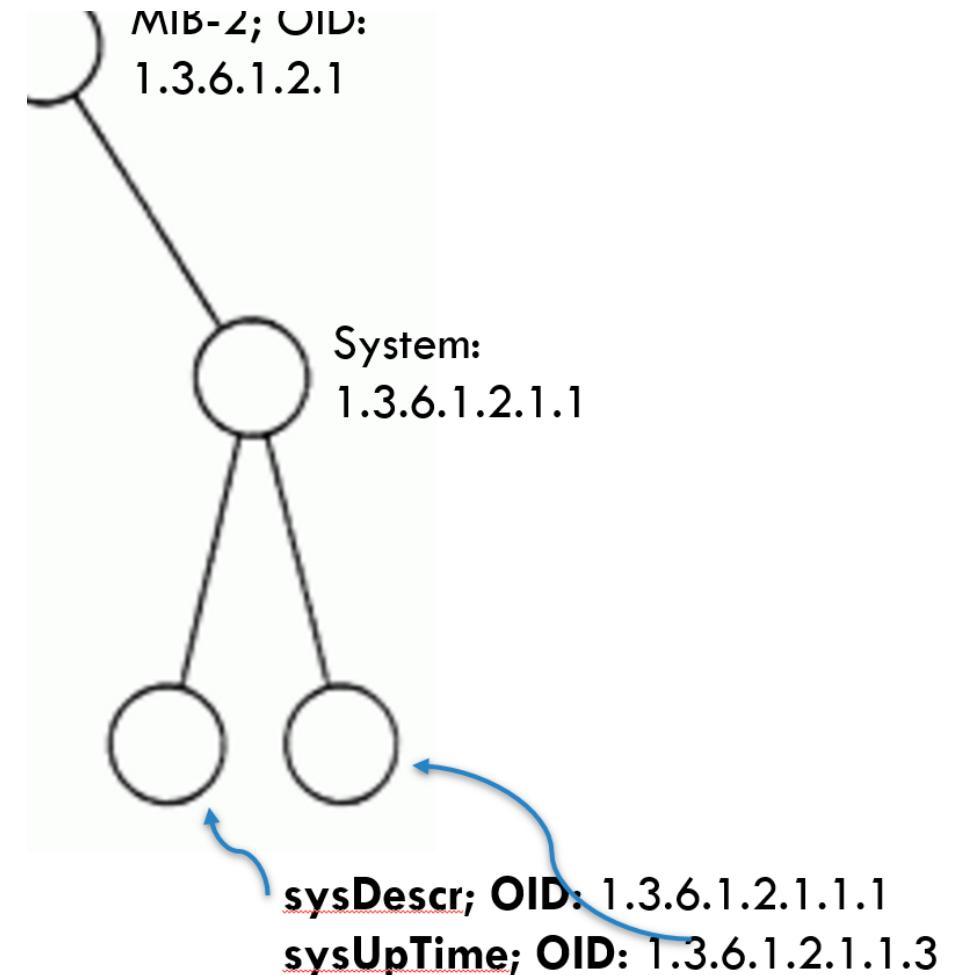
Interesujące nas dane zapisane szesnastkowo



MIKRO-TASK: WYKRYWANIE BŁĘDÓW

(MOGŁO BYĆ ZROBIONE WCZEŚNIEJ)

Dodajemy do naszego programu klasę zawierającą funkcję sprawdzającą poprawność zdekodowanego pakietu względem naszej wiedzy zawartej w drzewie MIB



EFEKT – TEST-CASE (PRZY SNMP-SET)

wejście



- 1) podajemy do naszego programu na wejście dane binarne reprezentujące zapytanie SNMP w formie szesnastkowej
- 2) wewnętrz program przekształca te dane na klasę zawierającą informacje (dekoduje dane)
- 3) program sprawdza czy dane zawarte w pakiecie odpowiadają typowi danych zawartemu w naszej bazie wiedzy (drzewo MIB)

wyjście



Program wyświetla informacje *z klasy*, oraz podaje informacje o niezgodności typów *lub* nieprawidłowych parametrach.

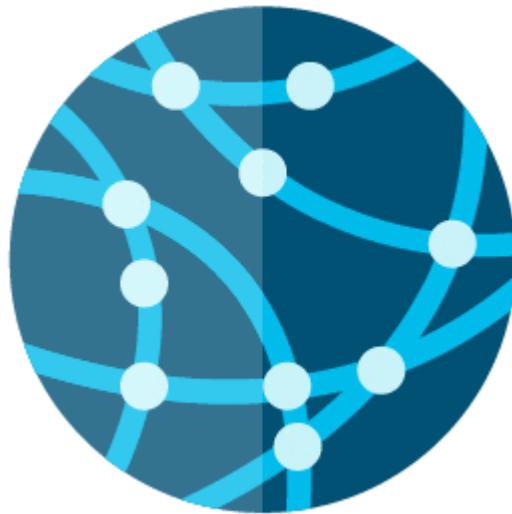
PUNKTY KONTROLNE



- ❖ Dekodowanie PDU z wektora bajtów
- ❖ Generowanie PDU o danym typie, i z danym RequestID
- ❖ Generowanie PDU z kodem błędu w przypadku jak dane w SNMP-SET nieprawidłowe (typ danych nie zgadza się z tym co w typie w liściu MIB o danym OID, czy też poza zakresem)
- ❖ Generowanie PDU zawierającego jedną wartość
- ❖ Generowanie PDU z wieloma polami OID/wartość

ETAP 5

FUNKCJONALNOŚĆ SIECIOWA



CEL

Celem jest implementacja klasy umożliwiającej:

- ❖ Odebranie danych z sieci wraz z ich wstępna weryfikacją (czy to nasz protokół? Czy hasło się zgadza? Czy udało się zdekodować ramkę?)
- ❖ Wywołanie ‚operational logic’/logiki biznesowej, w naszym przypadku będzie to zapis lub odczyt danych z systemu (na etapie bieżącego etapu funkcja to może nic nie robić), wygenerowanie raportu/rezultatu z logiki biznesowej (ramka statusu/błędu)
- ❖ Odesłanie rezultatu logiki biznesowej do klienta

Cecha	TCP	UDP
Skrót od	<i>Transmission Control Protocol</i>	<i>Universal Datagram Protocol</i>
Połączenie	Protokół połączeniowy	bezpołączeniowy
Porządkowanie kolejności pakietów	TAK	NIE
Prędkość	Wolniej	Szybciej
Niezawodność	Wysoka	Niska
Wielkość Nagłówka	20 bajtów	8 bajtów
Odczyt danych (Socket)	Jak ze Streamu	Jedno odczytanie zwróci wszystko co wysłane
Flow Control	Wykrywanie zatorów; kontrola prędkości przesyłu	Brak
Sprawdzanie błędów	Sprawdzanie poprawności danych, w razie potrzeby dane są przesyłane ponownie.	Sprawdzanie poprawności, ale dane niepoprawne są po prostu ignorowane.
Powiadomienia	TAK	NIE
Handshake	SYN, SYN-ACK, ACK	Brak
Obciążenie serwera	Większe	Mniejsze

WG. MODELU BERKLEY

SNMP

Protokół: UDP

Port: 161

WYSYŁANIE DANYCH

```
Socket socket = getSocket(type = „UDP”)
connect(socket, address = "1.2.3.4", port = „161”)
    send(socket, "Hello, world!")
    close(socket)
```

ODBIERANIE DANYCH

```
Socket socket = getSocket(type = „UDP”)
bind(socket, address = „INADDR_ANY”, port = „161”)
    receive(socket, bufor)
    close(socket)
```

PRZYKŁADOWY SERWER W C

```
int main(void) {  
    int sock; struct sockaddr_in sa; char buffer[1024]; ssize_t recsize;  
    socklen_t fromlen; //deklaracja potrzebnych struktur  
    memset(&sa, 0, sizeof sa);  
    sa.sin_family = AF_INET; //protokół IPv4  
    sa.sin_addr.s_addr = htonl(INADDR_ANY);  
    sa.sin_port = htons(161); //port 161  
    fromlen = sizeof sa; sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);  
    if (bind(sock, (struct sockaddr *) &sa, sizeof sa) == -1)  
    { perror(„gniazdo nie utworzone”); close(sock); exit(EXIT_FAILURE); }  
    for (;;) { recsize = recvfrom(sock, (void *)buffer, sizeof buffer, 0,  
        (struct sockaddr *) &sa, &fromlen);  
    if (recsize < 0)  
    {  
        //odczytanie danych nie powiodło się  
    }  
    //dane udało się odczytać
```

PRZYKŁADOWY Klient W C

```
int main(void) {
int sock; struct sockaddr_in sa; int bytes_sent; char
buffer[200]; strcpy(buffer, "hello world!");

sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sock == -1)
{ // nie udało się utworzyć gniazda}
memset(&sa, 0, sizeof sa);
sa.sin_family = AF_INET; /* Protokół to IPv4*/
sa.sin_addr.s_addr = inet_addr("127.0.0.1");
sa.sin_port = htons(161);
bytes_sent = sendto(sock, buffer, strlen(buffer), 0, (struct
sockaddr*)&sa, sizeof sa);
if (bytes_sent < 0) { //wysłanie danych nie powiodło się}
```

FUNKCJONALNOŚĆ SIECIOWA

SOCKET'y po stronie klienta:

```
#create an INET, STREAMing socket
s = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#now connect to the web server on port 80
# - the normal http port
s.connect(("www.mcmillan-inc.com", 80))
```

FUNKCJONALNOŚĆ SIECIOWA

SOCKET'y po stronie serwera (czyli np. po stronie agenta SNMP):

1

```
#create an INET, STREAMing socket
serversocket = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#bind the socket to a public host,
# and a well-known port
serversocket.bind((socket.gethostname(), 80))
#become a server socket
serversocket.listen(5)
```

2

```
while 1:
    #accept connections from outside
    (clientsocket, address) = serversocket.accept()
    #now do something with the clientsocket
    #in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

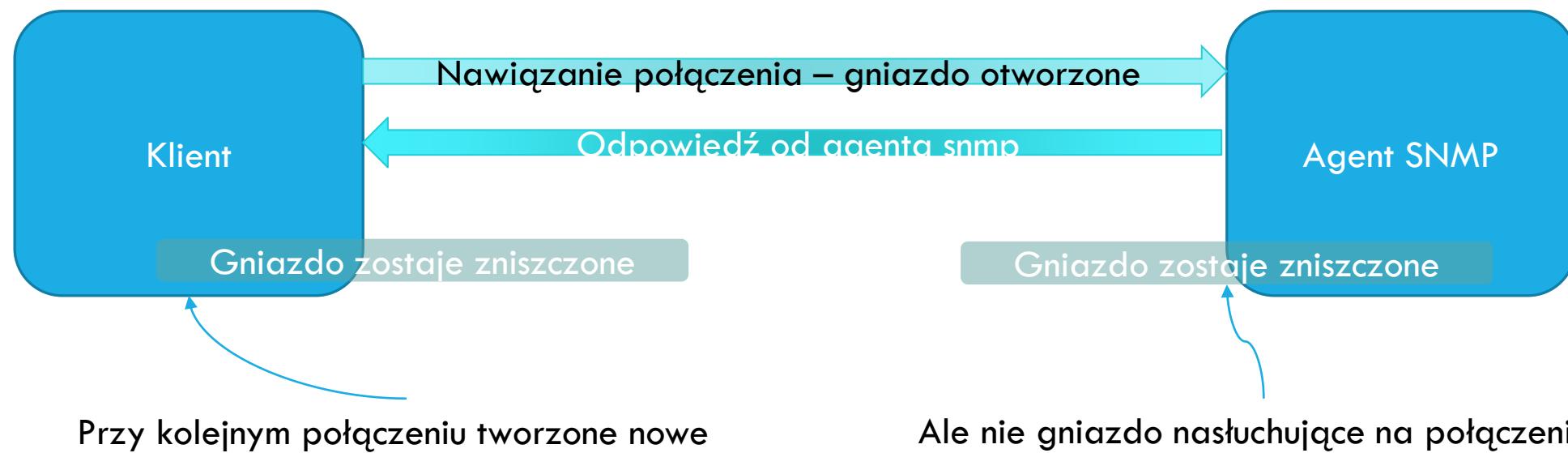
UROKI PROGRAMOWANIA SIECIOWEGO

1. WAŻNE: FLUSH po dokonaniu operacji zapisu do gniazda,
2. Jeśli korzystamy z buforowanego stream'u np. w Javie
3. Recv nie zawsze zwróci tyle danych z gniazda ile chcemy; może nie być dostępne w tym momencie tyle ile chcemy. Podajemy zawsze parametr ile możemy w tym momencie odebrać a nie tyle ile chcemy do Recv.
4. Recv czy send zwróci ilość bajtów które udało się obecnie przetworzyć. To nasza odpowiedzialność aby sprawdzić czy zostało przetworzone tyle ile chcieliśmy
5. Recv wykonyjemy tak długo aż nie będzie nic do odebrania

CHARAKTERYSTYKA POŁĄCZENIA SNMP

Podobnie jak HTTP

Gniazdo jest używane tylko dla jednego połączenia.



FUNKCJONALNOŚĆ SIECIOWA

1. Nasłuchujemy na porcie UDP nr. 161
2. Jeśli nadaje połączenie => akceptujemy

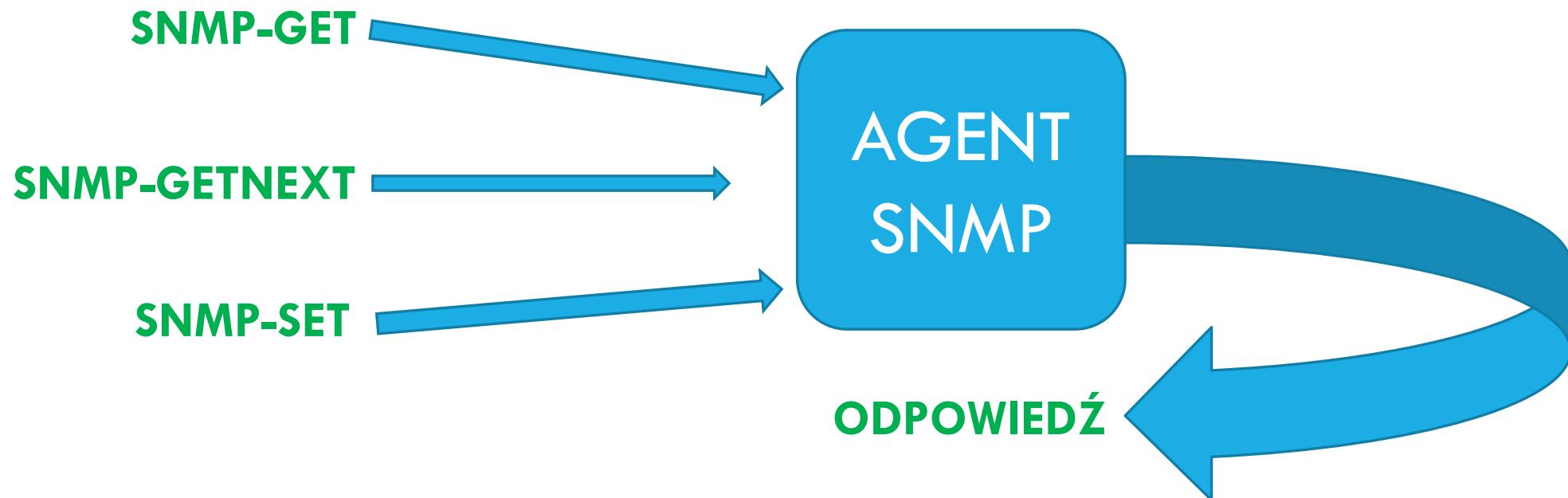
```
IPEndPoint ServerEndPoint= new IPPEndPoint(IPAddress.Any,9050);
Socket WinSocket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
WinSocket.Bind(ServerEndPoint);

Console.WriteLine("Waiting for client");
IPEndPoint sender = new IPPEndPoint(IPAddress.Any, 0)
EndPoint Remote = (EndPoint)(sender);
int recv = WinSocket.ReceiveFrom(data, ref Remote);
Console.WriteLine("Message received from {0}:", Remote.ToString());
Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
```

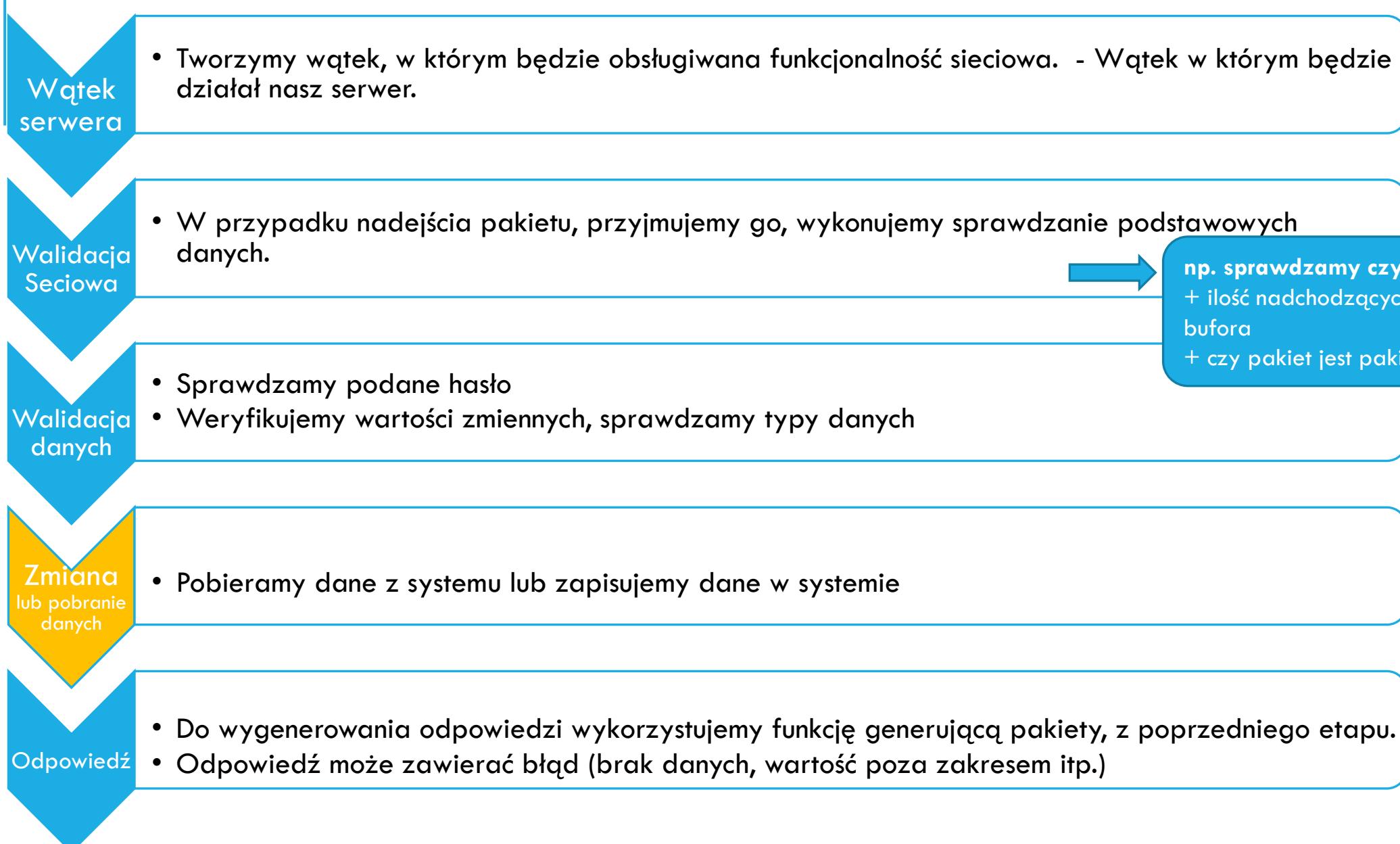
PUNKTY KONTROLE

- ❖ Deserializacja pakietów przy testowaniu z pakietem Net-SNMP
- ❖ Wygenerowanie poprawnej odpowiedzi i poprawna jej interpretacja przez jedno z poleceń pakietu Net-SNMP

TASK 5 – FUNKCJONALNOŚĆ SIECIOWA



TASK 5 – FUNKCJONALNOŚĆ SIECIOWA



(MIKRO) TASK 6 - OBSŁUGA ZAPYTAŃ SYSTEMÓWYCH

Tworzymy klasę umożliwiającą

1. odczytanie 3 wartości z Systemu operacyjnego
2. Zapisanie (wpłynięcie na) 3 wartości Systemu operacyjnego

FINAL TASK 7

Spinamy wszystkie poprzednie etapy w całość.

Test Scenario:

- 1) Program powinien móc odpowiedzieć prawidłowo na zapytanie SNMP-GET, SNMP-SET, oraz SNMP-GETNEXT
- 2) Wyłączamy z wymaganej funkcjonalności obsługę tabel (dla chętnych)
- 3) Po ustawieniu wartości przez SNMP-SET; wartość powinna zostać zapisana na stałe. Przetrwać restart programu
- 4) Program powinien być odporny na źle sformatowane pakiety (obsługa wyjątków) oraz wykrywać jeśli płynie zbyt dużo danych