AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA ELEKTROTECHNIKI I ELEKTROENERGETYKI

Praca dyplomowa magisterska

System wizualizacji danych medycznych DICOM z możliwością dostępu zdalnego

Autor: *Rafał Kobak*

Kierunek studiów: Elektrotechnika

Opiekun pracy: *dr inż. Paweł Turcza*

Kraków, 2015

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Rafał Kobak

Pragnę złożyć serdeczne podziękowania Panu dr inż. Pawłowi Turczy, za poświęcony czas, cenne wskazówki i pomoc w realizacji pracy.

Spis treści

1. Wstęp.

DICOM — Digital Imaging and Communications in Medicine — jest to międzynarodowy standard związany z obrazowaniem i przetwarzaniem diagnostycznych obrazów medycznych i powiązanych z nimi informacji. Format DICOM definiuje strukturę tych danych z zachowaniem, jakości niezbędnej do użytku klinicznego. Implementacja standardu DICOM jest wykorzystywana w niemal każdym urządzeniu obrazowania kardiologicznego, radiologicznego (aparaty do zdjęć rentgenowskich, tomografy komputerowe, rezonans magnetyczny). Standard DICOM znajduje również coraz szersze zastosowanie w urządzeniach wykorzystywanych w innych dziedzinach medycyny takich jak okulistyka czy stomatologia. Obecnie jest to najszerzej rozwijany standard opisu danych związany z opieką zdrowotną.

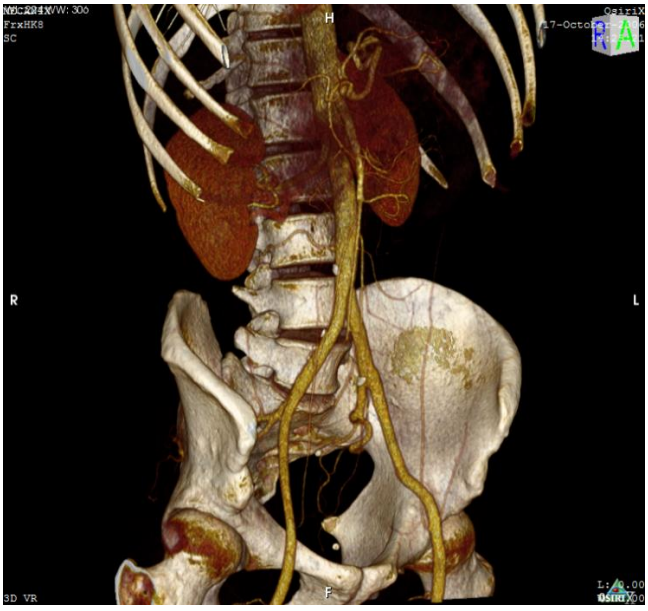
Celem pracy jest opracowanie systemu wizualizacji danych medycznych zapisanych w formacie DICOM. System składa się z aplikacji serwera realizującego operację wizualizacji i świadczącego usługi bazodanowe oraz aplikacji klienckiej działającej na urządzeniu mobilnym (tablet, smartfon), na którym będzie prezentowany wynik wizualizacji. Wizualizacja w postaci renderingu powierzchni jest zrealizowana z wykorzystaniem biblioteki VTK.

2. Standard DICOM.

2.1. Wprowadzenie.

DICOM – skrót od Digital Imaging and Communications in Medicine, czyli Obrazowanie Cyfrowe i Wymiana Obrazów w Medycynie jest to najbardziej uniwersalny oraz podstawowy standard stosowany w obrazowaniu medycznym. W obecnej postaci opracowany został w roku 1993 przez ACR/NEMA (American College of Radiology / National Electrical Manufacturers Association) dla potrzeb umożliwienia współpracy systemów używanych do wytwarzania, przetwarzania, interpretacji oraz przechowywania i transmisji danych medycznych reprezentujących lub związanych z obrazami diagnostycznymi w medycynie. Standard zawiera definicje formatu pliku oraz opis protokołu komunikacji sieciowej. Protokół komunikacyjny używa TCP/IP do komunikacji pomiędzy systemami. Dane w formacie DICOM mogę być wymieniane pomiędzy jednostkami zgodnymi z standardem.

Obrazy w formacie DICOM cechują się dużą objętością oraz wymagają specjalistycznego oprogramowania do obsługi. Ze względu na te dwie cechy do obsługi danych w formacie DICOM wymagany jest wysokiej, jakości sprzęt komputerowy oraz łącze o wysokiej przepustowości. Wykorzystanie technologii renderingu powierzchniowego umożliwia uzyskanie obrazów (Rys.2.1), które z powodzeniem mogą znaleźć zastosowanie również podczas zajęć dydaktycznych, zastępując rzeczywiste modele wykonane z tworzyw sztucznych.



Rys.2.1. Obraz w formacie DICOM z zastosowaniem renderingu powierzchniowego.

Standard DICOM znajduje szerokie zastosowanie w przetwarzaniu obrazów z urządzeń:

- Tomografii komputerowej (CT) – rys.2.2a
- Tomografii rezonansu magnetycznego (MRI)
- Pozytonowej tomografii emisyjnej (PET) – rys.2.2b
- Cyfrowej angiografii subtrakcyjnej (DSA)
- Radiografii konwencjonalnej (CR)
- Radiografii cyfrowej (DR)

Rys.2.2. To samo zdjęcie wykonane w technologii CT po lewe oraz PET po prawej.

Wykorzystywanych w takich dziadzinach medycyny jak:

- Kardiologii,
- Radiologii,
- Chirurgii,
- Neurologii,
- Stomatologii,
- Chirurgii,
- Onkologii,
- Okulistyki,
- Patologii,
- Weterynarii.

Z standardu DICOM korzysta większość systemów typu PACS (Picture archving and comunication system), czyli systemów archiwizacji obrazu i komunikacji.

2.2. Historia DICOM.

DICOM jest pierwszą wersją standardu rozwijanego przez ACR (American College of Radiology) i NEMA (National Electrical Manufacturers Association). Standard w obecnej postaci został opublikowany w roku 1993, jednakże jego początki sięgają znacznie wcześniej.

W latach 70-tych na wskutek intensywnego rozwoju technologii tomografii komputerowej oraz wzrostu ilości komputerów w zastosowaniach klinicznych, ACR oraz NEMA rozpoznało potrzebę stworzenia jednolitego standardu danych oraz transmisji dla medycznych obrazów diagnostycznych oraz powiązanymi z nimi danymi. Problemem był fakt, że każdy z producentów sprzętu medycznego stosował własny sposób opisu danych, który to często był trudny do zdekodowania przez radiologów oraz fizyków medycznych. Specjaliści potrzebowali tych danych np. do ustalenia poprawnej dawki promieniowania podczas radioterapii. Techniki interpretacji obrazów pochodzących z urządzeń medycznych różnych producentów były mocno zróżnicowane, co sprawiało wiele kłopotów i mogło prowadzić do pomyłek. Niezbędne, zatem okazało się opracowanie jednolitego standardu.

W roku 1983 ACR oraz NEMA połączyły siły i utworzyły komitet, którego celem miało być utworzenie takiego jednolitego standardu. Za cel postawiono sobie stworzenie standardu, który zapewniałby:

- Promowanie wykorzystania techniki cyfrowej w obrazowaniu medycznym,
- Ułatwienie rozwoju oraz możliwości rozszerzania PACS (Picture archving and communication system)
- Stworzenie bazy z danymi diagnostycznymi, oraz możliwości dostępu przez szeroką gammę urządzeń różnych producentów z różnych zakątków świata.

W roku 1985 a więc dwa lata po ustaleniu komitetu pojawiła się pierwsza wersja standardu pod nazwa ACR-NEMA Standard Publication No. 300-1985 opatrzona wersją 1.0. Nowy standard określił format danych, rodzaj transmisji oraz pierwszy słownik komunikatów. Bardzo szybko po ukazaniu, okazało się, że nowy standard zawierał wiele błędów oraz wewnętrznych niespójności. Potrzebne były liczne poprawki, których rezultatem było pojawienie się dwóch poprawek kolejno w sierpniu 1986 (No. 1) oraz w styczniu 1988 (No. 2).

W roku 1988 a więc 3 lata od ukazania się pierwszej wersji standardu ACR-NEMA opublikowana została druga wersja standardu – ACR-NEMA Standard Publication No. 300-1988 opatrzona wersją 2.0. Wersja ta zawierała wersje pierwszą wraz z obiema poprawkami oraz:

- Wsparcie dla urządzeń graficznych,
- Nowy schemat interpretowania obrazów,
- Nowe pola danych,
- Zdefiniowany sposób transmisji obrazu poprzez EIA-486.

Wersja druga standardu była znacznie lepiej dostosowana do współpracy z sprzętem medycznym. Pierwsza demonstracja standardu ACR-NEMA 2.0 miała miejsce w dniach 21-23 maja roku 1990 na uniwersytecie Georgetown. W wydarzeniu tym brały takie firmy jak:

- DeJarnette Research Systems,
- General Electric Medical Systems,
- Merge Technologies,
- Siemens Medical Systems,
- Vortech,
- 3M.

Sprzęt wykorzystujący standard ACR-NEMA 2.0 został zaprezentowany po raz pierwszy podczas corocznego spotkania RSNA (Radiological Society of North America) w roku 1990. Praktyczne zastosowanie pokazało, że wersja druga również nie jest wolna od błędów, konieczne były dalsze poprawki. Na wskutek tego powstało kilka niezależnych rozszerzeń standardu takich jak rozwijane przez University Hospital w Genewie Papyrus, czy stworzony przez Siemens Medical Systems i Philips Medical Systems SPI (Standard Product Interconnect).

Pierwsze wdrożenie standardu ACR-NEMA na szeroką skale miało miejsce w roku 1992 i zostało wykonane przez armię amerykańską (US Army) oraz amerykańskie siły powietrzne (US Air Force), jako część programu MDIS (Medical Diagnostic Imaging Support). Loral Aerospace oraz Siemens Medical Systems przewodziły konsorcjum firm, których celem było stworzenie pierwszego systemy PACS dla zastosowań militarnych. System został wdrożony we wszystkich liczących się oddziałach armii amerykańskiej oraz amerykańskich sił powietrznych.

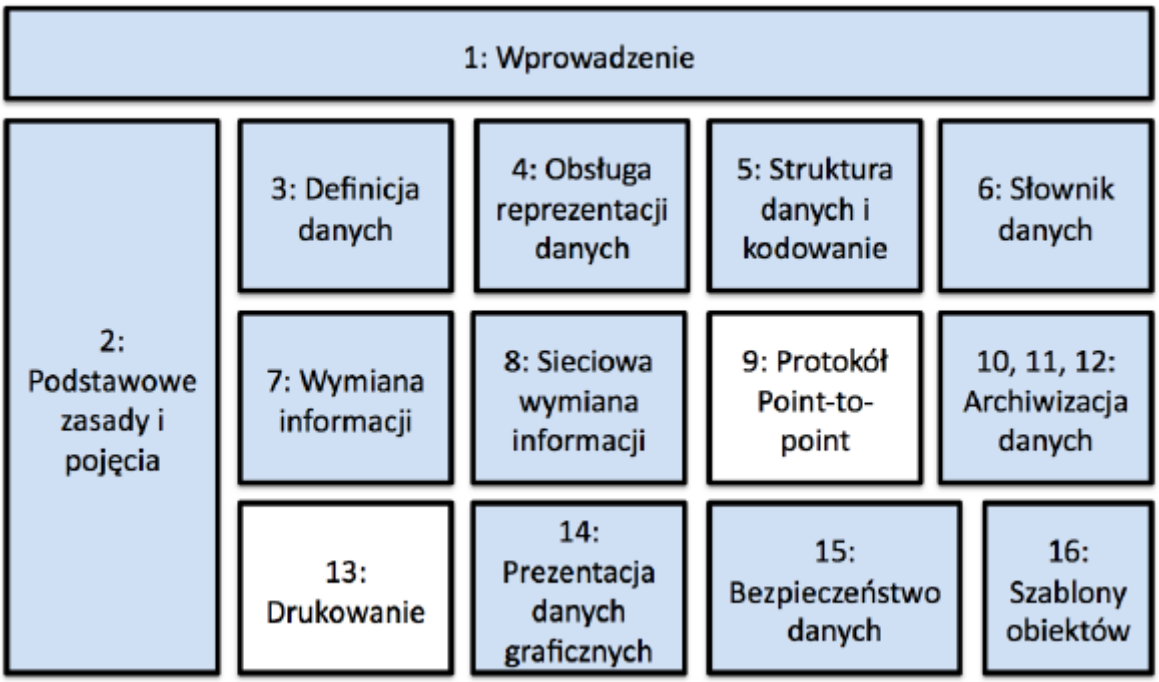
W roku 1993 ukazała się trzecia wersja standardu ACR-NEMA. Nazwa standardu została zmieniona na DICOM (Digital Imaging and Communications in Medicine).

Rys.2.3. Oficjalne logo standard DICOM.

W nowej wersji dodano nowe klasy, wsparcie dla obsługi sieci, oraz stworzono deklarację zgodności. Oficjalnie ACR-NEMA 3.0 czyli DICOM jest najnowszą wersją standardu, jednakże standard ten jest stale aktualizowany i rozszerzany. Kolejne aktualizacje oraz rozszerzenia oznacza się podając rok, w którym zostały dodane np. wersja 2007 standardu DICOM.

2.3. Dokumentacja standardu oraz model danych.

Standard DICOM zorganizowany jest w postaci wielostronicowego dokumentu podzielonego na poszczególne rozdziały. Najaktualniejszą wersję standardu można pozyskać z strony internetowej samego standardu DICOM to jest: http://dicom.nema.org/. Cały standard DICOM składa się z części przedstawionych na rys 2.4.



Rys.2.4. Części składowe (rozdziału) standardu DICOM.

Najbardziej istotne z punktu widzenia tematu pracy magisterskiej będą rozdziały 3 do 8 oraz 14. Każdy z nich zostanie przedstawiony w skrócie.

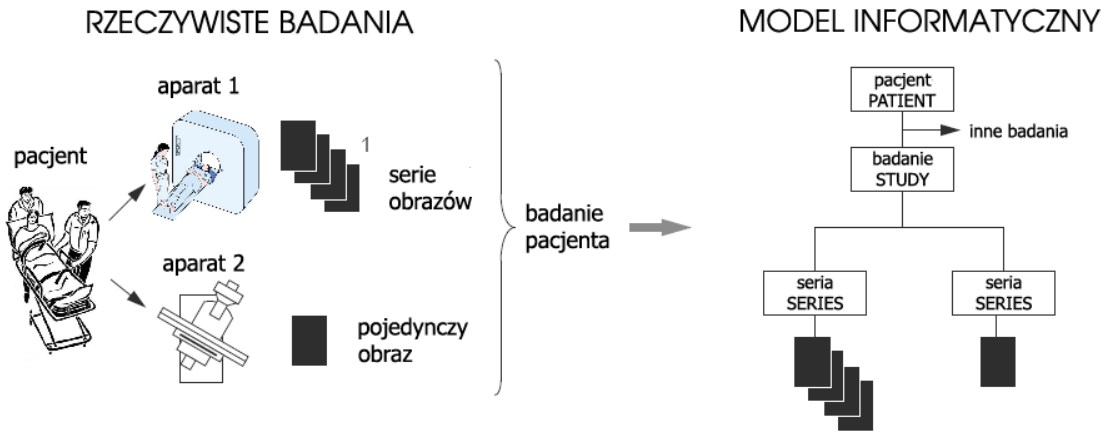
Rozdział trzeci standardu opisuje sposób definicji danych a więc określa między innymi ilość klas obiektów danych w skrócie IOC (Information Object Classes), które to zapewniają abstrakcyjne odzwierciedlenie rzeczywistych jednostek związanych z obrazowaniem medycznym takich jak dawka promieniowania, próbkowanie itp. Każda definicja IOC składa się z opisu tego, co dana klasa ma definiować oraz atrybutów definiujących daną wielkość. Atrybut składa się z nazwy oraz wartości, IOC nie zawierają wartości dla atrybutów. Wyróżnia się dwa typy IOC, zwyczajne oraz złożone.

Zwyczajne IOC zawierają tylko takie atrybuty, które są nieodłącznym elementem opisywanego przeze nie obiektu rzeczywistego. Dla przykładu Study IOC opisujący badanie, który w standardzie zdefiniowany jest, jako zwyczajny, zawiera atrybuty takie jak data badania (Study Date), czas badania (Study Time). Atrybuty te są nierozerwalnie związane z

każdym badaniem. Takie dane jak imię, nazwisko pacjenta nie są atrybutami Study IOC, jako, że są to atrybuty związane z pacjentem a nie bezpośrednio z badaniem.

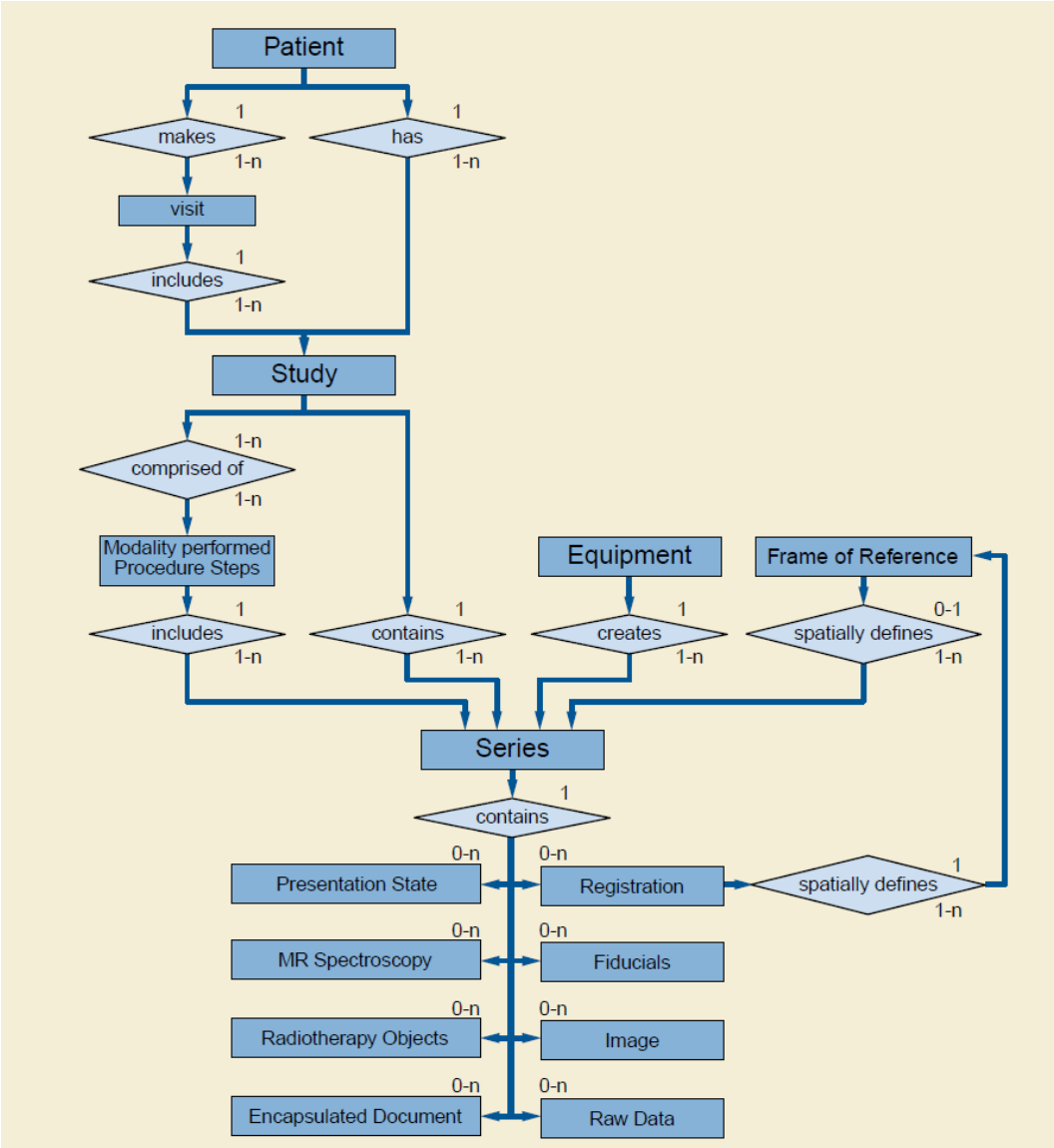
Złożone IOC mogą dodatkowo zawierać atrybuty pośrednio związane z samym obiektem rzeczywistym, który opisują. Na przykład IOC Computed Tomography Image opisujący zdjęcie z tomografu komputerowego, który zdefiniowany jest, jako złożony IOC, zawiera zarówno atrybutu bezpośrednio związane z rzeczywistym obiektem, takie jak data zdjęcia (Image Date) jak i atrybutu związane w sposób pośredni z opisywanym obiektem rzeczywistym takie jak imię pacjenta (Patient Name).

Dane w świecie rzeczywistym są ze sobą powiązane, tworzą pewien schemat powiązań rys.2.5. W przypadku danych medycznych pacjent może mieć na przykład badanie składające się z sesji na aparacie1 oraz aparacie2. Z aparatu1 może powstać pewna seria obrazów a z aparatu2 pojedynczy obraz. Pojedynczy obraz sam w sobie, nieosadzony w kontekście niewiele może powiedzieć. Liczy się to jak dany obraz jest powiązany z danym aparatem czy pacjentem. Budując model informatyczny należy mieć na uwadze, aby w sposób dokładny odwzorować te powiazania pomiędzy obiektami. Na rys.2.5. przedstawiono przykładowe rzeczywiste badanie oraz model informatyczny odpowiadający temu badaniu.



Rys.2.5. Badanie medyczne oraz odpowiadający badaniu model informatyczny.

Model danych medycznych wykorzystywany w standardzie DIOCM został przestawiony na rys.2.6. Model przedstawia sposób połączenia różnych informacji medycznych oraz zależności występujące pomiędzy nimi. Liczby obok strzałek reprezentują możliwą ilość połączeń pomiędzy poszczególnymi IOC.



Rys.2.6. Model rzeczywistych danym medycznych w standardzie DICOM.

Do najważniejszych informacji zawartych w takim modelu zaliczyć można:

- Dane pacjenta: imię i nazwisko, data urodzenia, data przyjęcia;
- Badania: elementy składowe badań, procedury, wyniki badań (raport);
- Serie danych: obrazy, dane surowe, tablice kolorów. Przykładem serii danych jest zestaw danych (slajdów) przedstawiających przekroje przez ciało pacjenta, otrzymane podczas rekonstrukcji danych CT dla konkretnych parametrów rekonstrukcji (np. rozdzielczość, odległość między przekrojami, filtr rekonstrukcji, czy parametry okna).

Rozdział czwarty standardu definiuje operacje przeprowadzane na obiektach danych opisanych w rozdziale trzecim. W rozdziale tym zdefiniowane są tak zwane klasy usług (Service classes). Klasa usług tworzy powiązania pomiędzy obiektami danych, tworzy operacje które mogą być wykonywane na obiektach danych. Przykładami klas usług są:

- Przechowywanie danych (Storage Service Class),
- Zapytania (Query Service Class),
- Zarządzanie drukowaniem (Print Management Class).

Rozdział piąty standardu DICOM określa jak aplikacje korzystające z DICOM będą konstruować zestawy danych (Data Sets) oraz w jaki sposób będą one zakodowane. Zestawy danych budowane są z klas obiektów danych oraz klas usług opisanych w rozdziałach trzecim i czwartym standardu. Zdefiniowany jest również sposób tworzenia strumieni danych przekazywanych w wiadomościach opisanych w rozdziale siódmym. Dodatkowo rozdział ten definiuje rodzaj technik kompresji obrazu jpeg zarówno stratnej jak i bezstratnej, oraz sposób kodowania znaków międzynarodowych.

Rozdział szósty poświęcony jest słownikom danych. Definiuje on zestawy danych DIOCOM (Dicom Data Elements) możliwe do wykorzystania podczas prezentacji informacji medycznych. Dla każdego takiego elementu rozdział szósty standardu definiuje jego unikalny znacznik składający się z grupy oraz numeru elementu, jego nazwę, typ wartości (całkowitoliczbowy, ciąg znaków itp), mnogość - jak wiele wartości może wystąpić na atrybut oraz czy jest oficjalnie wspierany czy może wycofywany wraz z każdą aktualizacją standardu.

Rozdział siódmy specyfikuje zarówno usługę jak i protokół wykorzystywany przez aplikację w środowisku obrazowania medycznego w celu wymiany informacji. Do wymiany informacji wykorzystuje się wiadomości. Wiadomości te składają się z strumienia rozkazów działającego na podobnej zasadzie jak strumień danych zdefiniowany w rozdziale piątym. Rozdział siódmy określa operacje i notyfikacje dostępne dla klas usług zdefiniowanych w rozdziale czwartym, zasady ustanawiania i kończenia połączeń sieciowych wyspecyfikowanych w rozdziale ósmym, zasady, które rządzą wymianą zapytań oraz odpowiedzi, zasady kodowania niezbędne do budowy wiadomości i strumieni rozkazów.

Rozdział ósmy poświęcony jest wymianie informacji poprzez sieć. Rozdział ten definiuje usługi komunikacyjne oraz protokoły wyższych warstw niezbędne przy pracy w środowisku sieciowym. Protokoły te oraz usługi mają na celu zapewnienie, że komunikacja sieciowa

pomiędzy aplikacjami DICOM przebiega w sposób zorganizowany i efektywny. Usługi komunikacyjne zdefiniowane w rozdziale ósmym są podzbiorem usług oferowanych przez standard ISO/OSI. Definicja usług komunikacyjnych wyższych warstw określa współprace wyższych warstw komunikacyjnych DIOCM w połączeniu z warstwą transportową protokołu TCP/IP.

W ostatnim wspomnianym rozdziale to jest rozdziale czternastym opisany jest sposób prezentacji danych graficznych. Rozdział ten definiuje zestandaryzowane funkcje dla spójnego wyświetlania obrazów w odcieniach szarości. Funkcje te zapewniają metody kalibracji dla poszczególnych systemów obrazowania. Celem jest zachowanie spójność dla różnych mediów prezentacji danych takich jak na przykład monitor i drukarka. Funkcje są zdefiniowane w oparciu o percepcję ludzkiego oka. Standard DICOM używa modelu Bartena opisującego percepcje ludzkiego oka.

2.4. Budowa plików w standardzie DICOM.

Informacje znajdujące się pliku DICOM podzielone są na dwie główne jednostki:

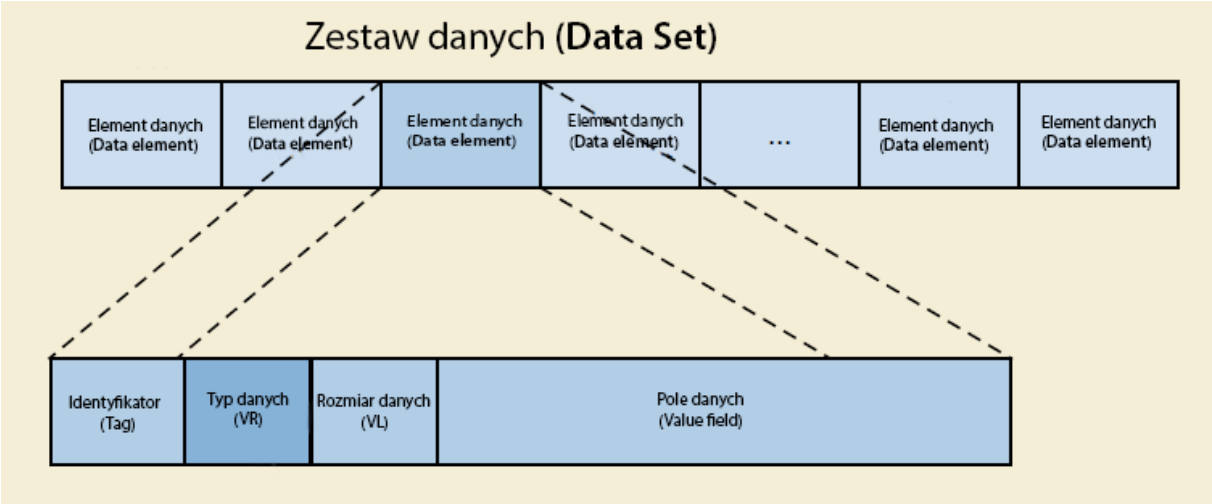
- Nagłówek, to jest część zawierającą informacje o pliku DICOM (Dicom-Meta-Information-Header),
- Dane obiektu (Dicom-Data-Set).

Nagłówek pliku zawierający informacje o pliku (Dicom-Meta-Information-Header) jest wymagany dla każdego pliku DICOM. Rozdział 10 standardu DICOM definiuje zawartość tej części pliku. Jednostka zawierająca dane (Dicom-Data-Set) przechowuje informacje o jednym obiekcie typu Service-Obiect-Pair instance (SOP instance). Obiektem tym może być np: pojedynczy przekrój z tomografii komputerowej (CT), rezonansu magnetycznego (MRI), czy opis zawartości nośnika, tak jak ma to miejsce w przypadku pliku DICOMDIR.

Podstawową jednostką danych w standardzie DIOCM jest zestaw danych (Data Set). Zestawy danych reprezentują instancje opisu rzeczywistego obiektu. Zestawy danych zbudowane są z tak zwanych elementów danych (Data Elements). Elementy danych zawierają zakodowane atrybuty oraz wartości dla opisywanych obiektów. Elementy danych w zestawie danych maja przypisany unikalny identyfikator (Data Element Tag). Elementy danych w zestawie danych, uporządkowane są względem identyfikatorów w sposób rosnący. W jednym zestawie danych może się znajdować tylko jeden element danych o danym identyfikatorze.

Budowa pojedynczego elementy danych przedstawiona została na rys 2.7. Element danych zbudowany jest z pól. Poniżej zostanie scharakteryzowane każde z nich:

- Identyfikator (Data Element Tag), pole składające się z dwóch liczb całkowitych określających grupę oraz element grupy, kilka przykładowych identyfikatorów wraz z opisem zebrano w tabeli tab.2.1;
- Typ danych (Value Representation), pole składające się z dwuznakowego stringu określającego sposób reprezentacji danych np. SS (Singed Short) oznacza 16 bitową liczbę całkowitą ze znakiem, a AS (Age String) wiek wyrażony w dniach (nnnD), tygodniach (nnnW), miesiącach (nnnM) bądź latach (nnnY), dostępne w standardzie DICOM typy danych zostały zebrane w tabeli 2.2;
- Rozmiar danych (Value Length), pole będące liczbą całkowitą bez znaku określające parzysta liczbę bajtów potrzebną do zapisania danych w polu danych;
- Pole danych (Value Field), pole z właściwymi danymi.



Rys.2.7. Struktura zestawów danych oraz elementów danych.

Wyróżnia się dwa typu elementów danych standardowe oraz prywatne. Standardowe elementy danych mają parzyste numery grupy wewnątrz identyfikatora natomiast prywatne nieparzyste.

Nie każdy element danych zdefiniowany jest w ten sam sposób, niektóre z nich nie zawierają części pól, bądź reprezentowane są one w inny sposób. Wyróżnia się trzy struktury, według których budowane mogą być elementy danych. Wszystkie trzy struktury zawierają pola identyfikatora, rozmiaru danych oraz pola danych. Pole typu danych jest polem, które różnicuje te trzy struktury, w jednej z nich pole to nie występuje w ogóle, natomiast w dwóch pozostałych jest obecne, ale z innych sposobem reprezentacji jego długości. W jednym zestawie danych nie mogą współistnieć elementy danych o różnej strukturze.

Tab.2.1. Przykładowe identyfikatory wraz z opisem.

Nazwa	**Identyfikator DICOM**	**Definicja**
Data badania *Study Date*	(0008,0020)	Data rozpoczęcia badania.
Producent *Manufacturer*	(0008,0070)	Producent urządzenia, które jest źródłem obrazu.
Opis badania *Study description*	(0008,1030)	Opis przeprowadzonego badania.
Płeć pacjenta *Patient Sex*	(0010,0040)	Płeć pacjenta zdefiniowana za pomocą typu wyliczeniowego, M dla mężczyzny, F dla kobiety oraz O - inna

Tab.2.2. Typy danych w standardzie DICOM

Typ danych	Skrót	Definicja	Rozmiar
Jednostka aplikacji *Application Entity*	AE	Łańcuch znaków definiujący jednostkę aplikacji, gdzie pierwszy jak i ostatni znak spacji nie jest znaczący. Wartości składające się wyłącznie ze spacji nie powinny być używane z tym typem.	maks. 16 B
Tekst wieku *Age String*	AS	Łańcuch znaków używający jednego z formatów – (nnnD, nnnW, nnnM, nnnY), gdzie nnn powinno zawierać liczbe dni dla D, tygodni dla W, miesięcy dla M oraz lat dla Y. Zapis „018M" reprezentuje wiek 18 miesięcy.	4 B
Znacznik atrybutu *Attribute Tag*	AT	Uporządkowana para 16-sto bitowych liczb całkowitych bez znaku, które są wartością pola identyfikatora w elemencie danych. Dla przykładu, pole identyfikatora o następującej postaci (0018,00FF) zostanie zakodowane, jako seria 4 bajtów, w zapisie Big-Endian, jako 00H, 18H, 00H, FFH natomiast w zapisie Little_Endian, jako 18H, 00H, FFH, 00H.	4 B
Tekst kodu *Code String*	CS	Łańcuch znaków z nieznaczącym pierwszym oraz ostatnim znakiem spacji.	maks. 16 B
Data *Date*	DA	Łańcuch znaków w formacie YYYYMMDD gdzie YYYY oznacza rok, MM miesiąc natomiast DD dzień miesiąca według kalendarza gregoriańskiego. Dla przykładu 20150714 oznacza 14 lipca roku 2015.	8 B
Tekst dziesiętny *Decimal String*	DS	Łańcuch znaków reprezentujący liczbę stało bądź zmiennoprzecinkową. Liczba stałoprzecinkowa powinna zawierać tylko znaki 0-9 z opcjonalnym znakiem '+/-' i opcjonalnym znakiem'.' będącym kropką dziesiętną. Liczba zmiennoprzecinkowa składa się dodatkowo ze znaku 'e' lub 'E' który to wskazuje początek eksponenty.	maks. 16 B
Czas i data *Date Time*	DT	Łańcuch znaków reprezentujący datę i czas w formacie YYYYMMDDHHMMSS.FFFFFF&ZZXX gdzie idąc od prawej do lewej: YYYY oznacza rok, MM miesiąc, DD dzień, HH godzinę (zakres od 0-23), MM minutę, SS sekundę. FFFFFF oznacza ułamkową część sekundy, &ZZXX jest opcjonalnym przyrostkiem dla przesunięcia czasu UTC gdzie & może przyjmować wartość +/- natomiast ZZ to godziny a XX minuty.	maks. 26 B
L. zmien. poj. precyzji *Floating Point Single*	FL	Liczba zmiennoprzecinkowa pojedynczej precyzji. Zapisywana jest w formacie 32 bitowym. Odpowiednik typu float z takich języków programowania jak C czy C++.	4 B
L. zmien. pod. precyzji *Floating point Double*	FD	Liczba zmiennoprzecinkowa podwójnej precyzji. Zapisywana jest w formacie 64 bitowym. Odpowiednik typu double z takich języków programowania jak C czy C++.	8 B
Liczba całkowita *Integer String*	IS	Łańcuch znaków reprezentujący liczbę całkowitą o podstawie 10. Typ ten powinien zawierać jedynie znaki 0-9 z opcjonalnym znakiem +/- na początku.	maks. 12 B

Tab.2.2. Typy danych w standardzie DICOM cd.

Długi łańcuch *Long string*	LO	Łańcuch znaków, który może zawierać znaki spacji z przodu jak i z tyłu. Typ nie może zawierać znaków sterujących z wyjątkiem ESC.	maks. 64 znaki
Długi tekst *Long Text*	LT	Łańcuch znaków, który może zawierać kilka akapitów. Typ ten może zawierać znaki graficzne oraz znaki sterujące oraz znaki spacji.	maks. 10240 znaki
Inny łańcuch bajtowy *Other Byte String*	OB	Łańcuch bajtów gdzie kodowanie zawartości jest zdefiniowane w składni przejść. Typ ten jest niewrażliwy na sposób zapisu (Little Endian/Big Endian)	def. w składni przejść.
Łańcuch słów binarnych *Other Double String*	OD	Łańcuch 64 bitowych słów binarnych.	2^{32} – 8 B
Łańcuch słów binarnych *Other Float String*	OF	Łańcuch 32 bitowych słów binarnych.	2^{32} – 4 B
Łańcuch słów binarnych *Other Words String*	OW	Łańcuch 16 bitowych słów binarnych.	def. w składni przejść.
Nazwisko Osoby *Person Name*	PN	Łańcuch znakowy zbudowany z 5 komponentów. Łańcuch może zawierać znaki spacji. Którykolwiek z pięciu komponentów może być pustym łańcuchem znaków.	maks. 64 znaki
Krótki łańcuch *Short String*	SH	Łańcuch znaków, który może zawierać znaki spacji. Łańcuch nie powinien zawierać znaków sterujących za wyjątkiem znaku ESC.	maks. 16 znaków
L. całko. długa ze znakiem *Signed Long*	SL	Liczba całkowita długa ze znakiem. Jest to liczba 32 bitowa, zakres wynosi: $-2^{31} \leq n \leq 2^{31} - 1$	4 B
Seria elementów *Sequence of Items*	SQ	Sekwencja grup elementów danych	-
L. całk. krótka ze zn. *Signed Short*	SS	Liczba całkowita ze znakiem o długości 16 bitów. Jej zakres to: $-2^{15} \leq n \leq 2^{15} - 1$	2 B
Krótki tekst *Short Text*	ST	Łańcuch znakowy, który może zawierać jeden lub więcej akapitów, znaki tekstowe oraz znaki sterujące.	maks. 1024 znaki
Czas *Time*	TM	Łańcuch znaków zapisany w konwencji HHMMSS.FFFFFF, gdzie HH to godziny, MM minuty, SS sekundy, natomiast FFFFFF reprezentuje ułamkowe części sekundy.	maks. 14 B

Tab.2.2. Typy danych w standardzie DICOM cd.

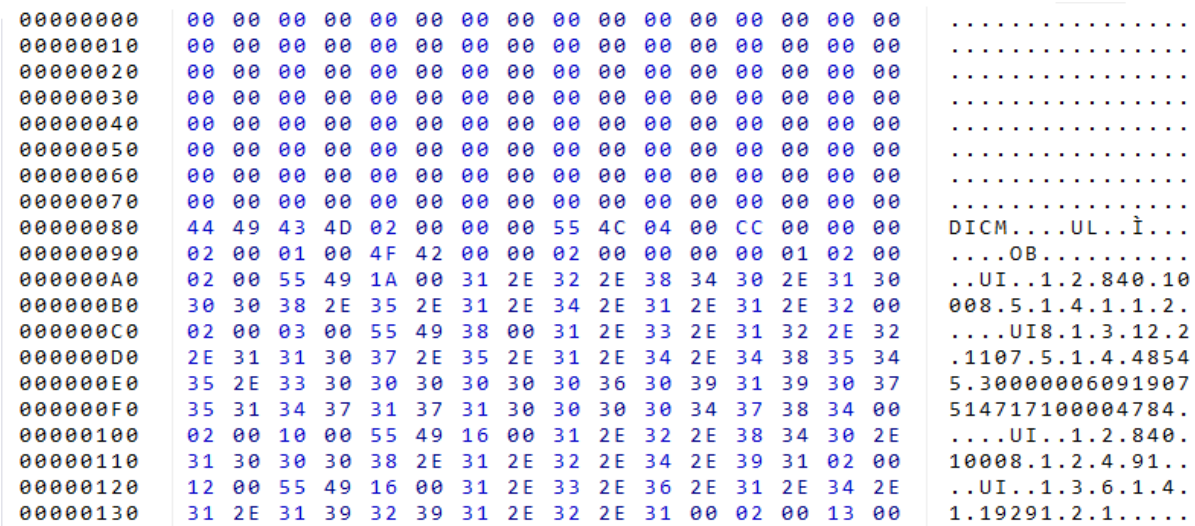
Nielimitowane znaki *Unlimited Characters*	UC	Łańcuch znaków, który może zawierać nieograniczoną długość znaków w tym spacji. Łańcuch nie powinien zawierać znaków sterujących za wyjątkiem znaku ESC.	maks. 2^{32} – 2 B
Unikalny identyfikator *Unique Identyfier*	UI	Łańcuch znakowy zwierający unikalny identyfikator. Identyfikator ten to seria komponentów liczbowych oddzielonych znakiem '.'.	maks. 64 B
L. całk. długa bez znaku Unsigned Long	UL	Liczba całkowita długa bez znaku. Długość tej liczby to 32 bity. Reprezentuje liczby z zakresu: $0 \leq n \leq 2^{32}$	4 B
Nieznany *Unknown*	UN	Łańcuch bitowy, w którym sposób kodowania znaków jest nieznany	różna
L. całk. krótka bez znaku *Unsigned Short*	US	Liczba całkowita krótka bez znaku. Długość tej liczby to 16 bitów. Reprezentuje liczby z zakresu: $0 \leq n \leq 2^{16}$	2 B
Nielimitowany tekst *Unlimited Text*	UT	Łańcuch znaków zawierający jeden bądź więcej akapitów, możne zawierać znaki graficzne i znaki sterujące.	maks. 2^{32} – 2 B

2.5. Zawartość binarna pliku DICOM.

Pliki w informatyce możemy podzielić na pliki tekstowe oraz pliki binarne. Pliki tekstowe to pliki zawierające dane zapisane w ustalonym formacie kodowania wraz ze znakami sterującymi, natomiast pliki binarne zawierają surowe dane zapisane w pamięci komputera bez przetwarzania na jakąkolwiek postać czytelną dla człowieka.

Pliki DICOM to pliki binarne. Dlatego też nie jest możliwy odczyt tego typu plików w edytorach tekstu takich jak Notatnik czy popularny Notepad++. Konieczne jest oprogramowanie umożliwiające podglądnięcie zawartości pliku binarnego tak zwany edytor heksadecymalny. Przykładem takiego oprogramowania jest HexEdit. HexEdit umożliwia obsługę plików o bardzo dużych rozmiarach – do 16 eksabajtów. Oprogramowanie pozwala na odczyt pliku i przeglądanie oraz edycję zawartych w nim danych. Edytor potrafi wyświetlać i modyfikować zawartość pliku wyświetlaną w postaci szesnastkowej, dziesiętnej, binarnej. Do odczytu zawartości binarnej pliku DICOM zostanie użyty właśnie program HexEdit.

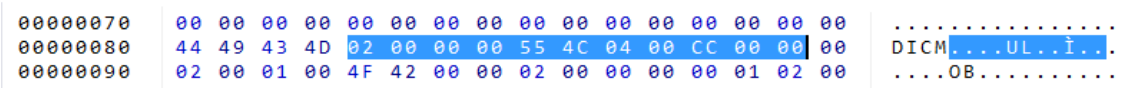
Poniżej na rys 2.8. przedstawiony został plik DICOM z rys 2.2. (wersja z tomografu komputerowego) odczytany w programie HexEdit - tryb szesnastkowy, słowo 8-bitowe.



Rys.2.8. Zawartość binarna pliku z rys.2.2.

Całość została zgrupowana w trzy kolumny, pierwsza reprezentuje adres w pamięci, druga zawartość pliku zapisaną heksadecymalnie, natomiast trzecia zawartość pliku przetłumaczona na ASCII.

Na przykładzie tego pliku przeanalizowany zostanie sposób interpretacji plików DICOM. W pliku przedstawionym na rys 2.8. można wyróżnić 128 bajtową preambułę pliku, zapełnioną zerami. Ta sekcja jest pusta. W dalszej części znajduje się ośmiu bajtowy identyfikator pliku w postaci 44 49 43 4D, który to po przetłumaczeniu na ASCII daje DICOM – identyfikator plików DICOM. W kolejnej sekcji znajduje się pierwszy zestaw danych (Data Set) składający się z kolejnych elementów (Data Element). Budowa zestawu danych została przestawiona na rys 2.7. w rozdziale 2.4. Pojedynczy element danych w odczytanym pliku został uwidoczniony zakreśleniem na rys 2.9.



Rys.2.9. Pojedynczy Data Element w pliku DICOM.

Pierwsze cztery zakreślone bajty to tak zwany identyfikator elementu (Tag), składający się z dwubajtowego identyfikatora grupy oraz dwubajtowego identyfikatora elementu grupy. Kolejne dwa bajty to kod typu danych (VR – value representation), dla zaznaczonego fragmentu jest to jest to UL, czyli Unsigned Long - 32 bitowa liczba całkowita bez znaku. W zależności od VR kolejno następuje od dwóch do dziesięciu bajtów przeznaczonych na długość danych, dla zaznaczonego fragmentu są to 4 bajty. Pozostałe bajty zaznaczonego fragmentu to obszar z danymi. Dalej następują kolejne pola elementów (Data Element) zakodowane w analogiczny sposób. Znajomość binarnej budowy pliku DICOM będzie niezbędna do jego prawidłowego przetwarzania.

2.6. Podział standardu DICOM.

Standard DICOM jest wykorzystywany w różnych gałęziach medycyny. Jest ze względu na ten fakt bardzo rozbudowanym standardem, przez co konieczne stało się podzielenie go na pomniejsze wyspecjalizowane grupy robocze (work groups) skojarzone z konkretna węższa dziedziną. Poniżej zamieszczone jest zestawienie każdej z tych grup roboczych z jednozdaniowym opisem:

- WG-01 Informacje o sercu i naczyniach krwionośnych (Cardiac and Vascular Information), zajmuje się wymiana informacji w dziedzinie układów naczyniowych, mocno współpracuje z drugą oraz ósmą grupą roboczą;
- WG-02 Projekcja radiografii i angiografii (Projection Radiography and Angiography), utrzymanie oraz rozwój obiektów XA, XRF, DX, CF zarówno w 2D jak i 3D związanych z radiografią;
- WG-03 Medycyna nuklearna (Nuclear Medicine), rozwój w dziedzinie wymiany informacji w obrazowaniu PET;
- WG-04 Kompresja (Compression), opracowanie sposobów kompresji obiektów standardu DICOM – obecnie dostępne (JPEG, RLE, JPEG-LS, JPEG2000, JPGPIP);
- WG-05 Nośniki wymiany danych (Exchange Media), rozwój w dziedzinie nośników wymiany danych, nośniki wykorzystywane w PACS;
- WG-06 Podstawa standardu (Base Standard), utrzymanie spójności całego standardu, publikacje nowych odsłon w porozumieniu z NEMA;
- WG-07 Radioterapia (Radioteraphy), rozwój i utrzymanie obiektów wykorzystywanych w urządzeniach radioterapii, opracowywanie obiektów dla nowych sposobów leczenia;
- WG-08 Strukturyzacja raportów (Structured Reporting), zajmuje się rozwojem norm, definicja nowych szablonów w standardzie DICOM;
- WG-09 Okulistyka (Ophthalmology), przygotowanie oraz utrzymywanie i rozwój obiektów wykorzystywanych w okulistyce;
- WG-10 Doradztwo strategiczne (Strategic Advisory), opracowuje strategie rozwoju standardu DIOCOM, długofalowe cele, współpraca z innymi organizacjami;

- WG-11 Standardy wyświetlania (Display Function Standard), rozwój sposobów wyświetlania danych zgodnie z standardem, opracowanie obiektów;
- WG-12 Ultrasonografia (Ultrasound), sprostanie wymaganiom stawianym przez ultrasonografie, opracowywanie obiektów 3D/4D, funkcje pomiaru;
- WG-13 Światło widzialne (Visible Light), rozwój standardu w zakresie stałego i ruchomego światła widzialnego, generowanego przez urządzenia wykorzystywane w medycynie takie jak endoskopy, mikroskopy;
- WG-14 Bezpieczeństwo (Security), rozwój standardu w zakresie bezpieczeństwa wymiany informacji, opracowywanie nowych sposobów kodowania;
- WG-15 Mammografia i CAD (Mammography and CAD), opracowywanie i rozwój obiektów wykorzystywanych w obrazowaniu piersi, opracowywanie struktury wyników badań z wykorzystaniem CAD;
- WG-16 Rezonans magnetyczny (Magnetic Resonance), dostosowywanie obiektów standardu DICOM do nowych urządzeń oraz technologii w zakresie obrazowanie z wykorzystaniem rezonansu magnetycznego;
- WG-17 3D, rozwój standardu dla współpracy z obrazami 3D oraz innymi wielowymiarowymi strukturami danych;
- WG-18 Badania kliniczne i edukacja (Clinical Trials and Education), praca nad rozszerzaniem standardu dla nowych rodzajów badań klinicznych z wykorzystaniem obrazowania;
- WG-19 Dermatologia (Dermatology), obrazowanie w dziedzinie dermatologii, według informacji z oficjalnej strony standardu zespół tymczasowo nieaktywny;
- WG-20 Integracja pomiędzy standardami obrazowania medycznego (Integration of Imaging and Information Systems), opracowanie metod zapewnienia spójność z innymi standardami np. HL7;
- WG-21 Tomografia komputerowa (Computed Tomography), dostosowywanie obiektów standardu DICOM do nowych urządzeń oraz technologii w zakresie obrazowania z wykorzystaniem tomografii komputerowej;
- WG-22 Stomatologia (Dentistry), opracowanie oraz utrzymanie obiektów związanych z stomatologią, rozwój standardu DICOM w aspektach związanych z symulacją leczenia, poprzez wykorzystanie projektowania wspomaganego komputerowo;

- WG-23 Hosting aplikacji (Application Hosting), opracowanie metod ujednolicenia współpracy pomiędzy oprogramowaniem klienta i serwera;
- WG-24 DICOM w chirurgii (DICOM in Surgery), rozwój standard w kierunku zdalnych operacji chirirgicznych;
- WG-25 Weterynaria (Veterinary Medicine), rozwój obiektów standardu dla zastosowań weterynaryjnych;
- WG-26 Patologia (Pathology), zapewnienie obsługi obrazów wykorzystywanych w patologii w tym autopsji;
- WG-27 Technologie Internetowe (Web Technology for DICOM), wykorzystanie technologii internetowych dla stworzenia rozszerzeń standardu w zakresie dystrybucji obrazów poprzez sieć Internet;
- WG-28 Fizyka (Physics Strategy), rozwija te elementu standardu, które wymagają specjalistycznej wiedzy w zakresie fizyki medycznej;
- WG-29 Edukacja, komunikacja i popularyzacja (Education, Communication, and Outreach), promowanie standard DIOCM, edukowanie na temat korzyści wynikających z korzystanie z standardu, organizacja konferencji, poszukiwanie osób mogących wspomóc rozwój standardu;
- WG-30 Obrazowanie domowych zwierząt (Small Animal Imaging), obrazowanie w weterynarii dla zwierząt domowych, opracowanie obiektów standradu, najnowsza grupa robocza;

Szczegółowe informacje na temat każdej grupy roboczej, takie jak adres, osoba koordynująca, obecne krótko i długofalowe cele znaleźć na oficjalnej stronie standardu w pliku opisującym strategie: http://medical.nema.org/dicom/geninfo/strategy.pdf

2.7. Oprogramowanie wykorzystujące standard DICOM.

Powstało kilka implementacji standardu DICOM zarówno w postaci bibliotek kompatybilnych z różnymi językami programowania jak i w postaci gotowych aplikacji do obsługi plików w standardzie DICOM. Przykładami bibliotek są:

- DCMTK – DICOM Toolkit;
- gdcm – Grassroots DICOM.

DCMTK jest zestawem bibliotek i aplikacji implementujących dużą cześć standardu DICOM. Udostępnia aplikacje do odczytywania, tworzenia i konwertowania plików DICOM, komunikacji poprzez sieć Internet. DCMTK napisane jest po części w ANSI C i w C++. Rozprowadzany jest na licencji wolnego oprogramowania. Biblioteki mogą być skompilowane po systemami Windows, Linux oraz MacOS. Szczegółowe informacje oraz kod źródłowy dostępne są pod adresem - http://dicom.offis.de/dcmtk

Biblioteka gdcm (Grassroots DICOM) jest kolejną implementacją standardu DICOM. Została zaprojektowana, jako biblioteka open-source w celu umożliwienia badaczom bezpośredni dostęp do danych DICOM. Biblioteka gdcm zawiera definicję formatu pliku oraz protokół komunikacji sieciowej. Zachowana jest kompatybilność z wcześniejszymi wersjami standardu to jest ACR-NEMA 1.0 oraz 2.0. Napisana jest w języku C++, jednak ma zdefiniowane wrappery dla innych języków programowania takich jak Python, Java oraz C#. Prowadzone są prace nad przygotowaniem wrapperów dla jeżyków Perl oraz PHP. Biblioteka usiłuje zapewnić wspieracie dla wszystkich dostępnych w DICOM formatów zapisu obrazów. Wsparcie zostało zapewnione dla takich formatów jak:

- RAW,
- JPEG kompresja stratna wersja 8 i 12 bitowa,
- JPEG kompresja bezstratna wersja 8-16 bitowa,
- JPEG 2000,
- RLE,
- JPEG-LS.

Kod źródłowy biblioteki dostępny jest pod adresem http://sourceforge.net/projects/gdcm/ natomiast pod adresem http://gdcm.sourceforge.net/wiki/ znajduje się wirtualna encyklopedia na temat biblioteki.

Powstały również gotowe aplikacje do obsługi zdjęć medycznych zapisanych w formacie DICOM. Jeśli chodzi o darmowe rozwiązania warto wspomnieć o programach:

- MicroDicom, przeglądarka i nie tylko obrazów medycznych w formacie DICOM – strona domowa projektu http://www.microdicom.com/
- OsiriX(Lite), kolejna przeglądarka plików DIOCM, kompatybilna jedynie z systemami MacOSX, strona domowa projektu http://www.osirix-viewer.com/

Z rozwiązań komercyjnych warto wspomnieć o następujących aplikacjach:

- OsiriX –kompleksowe środowisko do obsługi zdjęć medycznych w formacie DICOM oraz do komunikacji z systemami PACS, kompatybilna jedynie z systemami MacOSX, strona domowa projektu http://www.osirix-viewer.com/
- rsr2 – polska przeglądarka plików DICOM, strona domowa https://rsr2.pl/

Warta uwagi jest aplikacja telemedyczna TeleDICOM stworzona przez katedrę informatyki Akademii Górniczo-Hutniczej w Krakowie. Jest to aplikacja przeznaczona dla lekarzy medycyny, umożliwia wzajemną konsultacje wyników badań medycznych pomiędzy lekarzami różnych często bardzo wąskich specjalizacji pomimo geograficznego oddalenia. Aplikacja może być również wykorzystywane w edukacji studentów medycyny oraz wspierać organizowanie konferencji naukowych, jak i wspierać ciągły rozwój zawodowy lekarzy. Strona domowa projektu http://www.teledicom.pl/index.php/pl/

3. System archiwizacji obrazu i komunikacji.

PACS, czyli system archiwizacji obrazu i komunikacji, jest to technologia zapewniająca ergonomiczny sposób składowania oraz wygodny dostęp do obrazów z różnych źródeł wykorzystywanych w obrazowaniu medycznym. Formatem wykorzystywanym w PACS jest opisany w rozdziale drugim format DICOM. Inne dane mogą być przechowywane w innych formatach jak na przykład PDF.

System PACS składa się z następujących elementów:

- Urządzeń, które są źródłem cyfrowych zdjęć medycznych (rentgen, rezonans komputerowy),
- Oprogramowania przetwarzające zdjęcia pochodzące ze źródeł,
- Bazy danych z obsługą zdjęć cyfrowych,
- Podsystemu umożliwiającego wprowadzanie danych pacjentów,
- Podsystemu odpowiedzialnego za archiwizację zdjęć cyfrowych,
- Podsystemu umożliwiającego komunikację to znaczy przesyłanie zdjęć cyfrowych pomiędzy użytkownikami,
- Podsystem zapewniającego bezpieczeństwo danych przed nieuprawnionym dostępem.

Ideą systemu PACS została po raz pierwszy przedstawiona na spotkaniu radiologów w roku 1982. We wczesnych latach dziewięćdziesiątych, pierwszy udany system tego typu został wdrożony w szpitalu Hammersmith za sprawą działań doktora Harolda Glassa. Szpital ten, jako pierwszy w Wielkiej Brytanii całkowicie zrezygnował z przechowywania starych zdjęć na kliszach fotograficznych. Wcześniej w roku 1982 na uniwersytecie w Kansas podjęto próbę instalacji tego typu systemu, jednak nie zakończyła się ona sukcesem.

System PACS znalazł wiele zastosowań, głównym z nich jest zastąpienie tak zwanych „twardych kopii", dzięki PACS nie ma potrzeby przechowywania klisz fotograficznych ze zdjęciami medycznymi. Zapewnia to oszczędność fizycznego miejsca oraz zmniejszenie czasu dostępu do tego typu danych. Korzystając z PACS uzyskuje się dostęp zdalny do danych medycznych z dowolnego miejsca na świecie, otwiera to także możliwości dla przeprowadzania teleradiologi wśród lekarzy znajdujących się w różnych miejscach. Dzięki PACS możliwa jest także archiwizacja obrazów i przechowywanie ich, jako miękkich kopii. Dodatkowym atutem jest również automatyzacja procesów, obrazy medyczne mogą być przesyłane z urządzeń źródłowych do systemu w sposób automatyczny. System zapewnia

również interfejs do integracji z innymi systemami wykorzystywanymi w szpitalach takimi jak Szpitalny System Informatyczny (HIS), czy Radiologiczny System Informatyczny (RIS).

Fizycznie infrastruktura PACS składa się z 5 klas systemów komputerowych połączonych ze sobą poprzez sieć. Są to radiologiczne systemy obrazowania, komputery zbierające wyniki badań, kontrolery klastrów, przeglądowe stacje robocze oraz serwery baz danych. Kluczowa jest odpowiednia architektura systemu ze względu na wydajność całego systemu. Architektura ta powinna uwzględniać trzy poziomy modelowania. Pierwszy poziom jest zewnętrznym modelem danych, tworzonym z punktu widzenia lokalnego użytkownika. W skład tego modelu wchodzą opis transakcji, powiązane obiekty, priorytety transakcji, częstotliwość żądań oraz wymagane zasoby. Drugi poziom jest to model koncepcyjny, który na podstawie tych danych tworzy zintegrowana strukturę danych wspieraną przez każdą aplikację PACS. Trzeci ostatni poziom, jest modelem uwzględniającym fizyczna implementację, która musi w sposób logiczny zaimplementować model koncepcyjny przy zachowaniu odpowiedniej niezawodności i wydajności. W modelu tym występują duże (zdjęcia) i małe (teksty) jednostki danych, zarządzane w inny sposób. Do tych mniejszych jednostek wykorzystywane są zazwyczaj komercyjne relacyjne bazy danych natomiast dla tych większych wykorzystywane są specjalne przeglądowe stacje robocze. Powszechnym zaleceniem jest, aby podczas tworzenia systemu PACS korzystać z popularnych i standaryzowanych rozwiązań. Jeśli chodzi o system operacyjny powinno wykorzystywać się system Unix, Linux, Windows lub MacOS, jeśli chodzi o protokół transmisyjny to preferowanym jest TCP/IP, system bazodanowy SQL, standard obrazów medycznych – DICOM.

Komunikacja w serwerze PACS jest zrobiona z wykorzystaniem komunikatów DIOCM, mających analogiczną budowę do nagłówków obrazów DICOM. Różnicą jest fakt, że używa się innych atrybutów. Sposób komunikacji zostanie przedstawiony na przykładzie zapytania C-FIND. Zapytanie to realizowane jest w następujący sposób:

1. Klient nawiązuje połączenie z serwerem PACS.
2. Klient buduje na podstawie atrybutów DICOM, pustą wiadomość C-FIND.
3. Klient wypełnia przygotowaną wiadomość kluczami dopasowanymi według np. numeru identyfikacyjnego pacjenta.
4. Klient wypełnia zerami wszystkie atrybuty, co, do których chce wydobyć informacje z serwera.
5. Klient wysyła tak wypełniona wiadomość do serwera.
6. Serwer odsyła do klienta listę odpowiedzi C-FIND będącą również listą atrybutów DICOM.

7. Klient przetwarza odebrana wiadomość i wydobywa z niej listę atrybutów, którymi jest zainteresowany.

Systemy PACS są w coraz większym stopniu wykorzystywane w coraz większej liczbie placówek medycznych, również tych mniejszych. Dostępne są gotowe rozwiązania, zarówno komercyjne jak i bezpłatne. Przykładami płatnych rozwiązań są System Eskulap oraz Medinet PACS, poniżej pokrótce opisany zostanie system Eskulap.

Eskulap jest to zintegrowany system informatyczny szpitala, platforma technologiczna przeznaczona dla sektora opieki zdrowotnej. System stworzony został przez pracowników politechniki poznańskiej. Moduł PACS systemu pracuje na systemach Windows oraz Linux, obsługuję szeroką gamę plików DICOM, nieskompresowane w konwencji little i big endian, skompresowane za pomocą algorytmów rle, jpeg. Umożliwia przenoszenie obrazów na nośniki archiwalne oraz współprace z drukarkami DICOM. Szczegółowe informacje na temat systemu można znaleźć pod adresem: https://www.systemeskulap.pl/oferta/pacs/

4. Struktura systemu wizualizacji danych medycznych DICOM.

4.1. Architektura oraz podstawowe założenia.

System wizualizacji danych medycznych będący tematem niniejszej pracy składa się z dwóch zasadniczych części. Częściami tymi są aplikacja serwera działająca pod kontrola dowolnego systemu operacyjnego z grupy systemów linux oraz aplikacja kliencka działająca pod kontrolą systemu android. W założeniu działania systemu aktywna jest jedna aplikacja serwera oraz wiele aplikacji klienckich. Aplikacje komunikują się ze sobą za pomocą WiFi z wykorzystaniem protokołu TCP/IP. Ogólny schemat systemu przedstawiony został na rys.4.1.

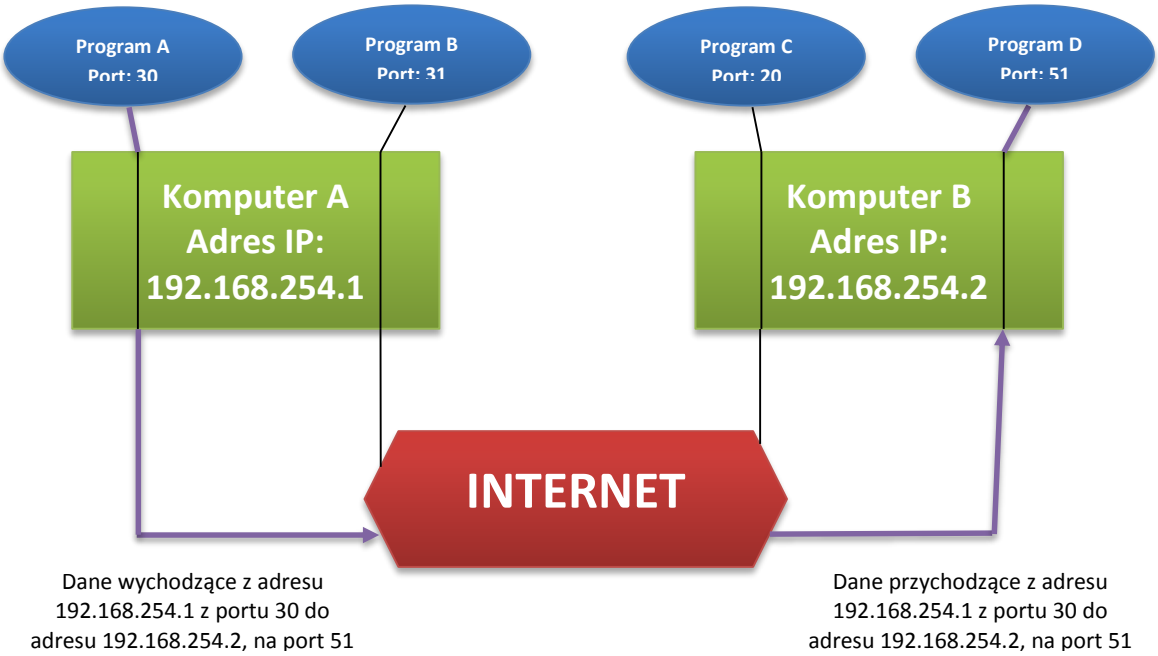
Rys.4.1. Ogólny schemat systemu wizualizacji danych medycznych DICOM.

Założeniem jest, aby to aplikacja serwera była odpowiedzialna za operacje wymagające dużych nakładów pracy procesora czy wysokiego zużycia pamięci systemowej RAM, natomiast aplikacja kliencka była możliwie lekka. Założenie podyktowane jest tym, iż aplikacja serwera w założeniu ma działać na komputerze stacjonarnym lub laptopie, które cechują się względnie wysoką mocą obliczeniową oraz możliwością zasilania z sieci energetycznej natomiast aplikacja kliencka ma działać na tabletach czy smartfonach, których moc obliczeniowa jest znacznie niższa oraz co równie istotne działają z wykorzystaniem baterii lub akumulatorów, co powoduje, że zbytnie obciążenie przekładałoby się negatywnie na długość użytkowania urządzenia bez konieczności ładowania. Zadania aplikacji serwera oraz aplikacji klienta przestawione zostaną w rozdziałach 5 oraz 6.

4.2. Programowanie sieciowe.

Medium komunikacyjnym pomiędzy aplikacja serwera oraz aplikacjami klientów jak zostało wspomniane jest protokół TCP/IP. Obsługa tego protokołu z poziomu aplikacji wymaga znajomości podstaw programowania sieciowego.

Połączenie sieciowe związane jest z takimi pojęciami jak adres IP, numer portu. Uproszczony schemat połączenia sieciowego przedstawiono na rys.4.2. Adresem IP jest 32-bitowa liczba bez znaku, która jednoznacznie identyfikuje komputer w sieci. Przykładem adresu IP jest 192.168.254.1. Adres IP nadawcy jest informacją dla odbiorcy, z którego komputera wysłano dane, jednak nic nie mówi o tym, który program na komputerze nadawcy je wysłał. Podobnie w drugą stronę, adres IP odbiorcy pozwala określić nadawcy komputer, który ma otrzymać dane jednak nie konkretny program. Identyfikatorem konkretnej aplikacji zarówno po stronie nadawcy jak i odbiorcy jest numer portu. Numer portu jest to 16-bitowa liczba bez znaku, na przykład port 22 zarezerwowany dla protokołu ssh (*secure shell*). Liczba ta jednoznacznie identyfikuje połącznie w obrębie danego adresu IP. Dlatego konkretne połączenie sieciowe jest identyfikowane poprzez dwie pary adresów IP i numerów portu, jedna para po stronie nadawcy, druga po stronie odbiorcy.



Rys. 4.2. Uproszczony schemat połączenia sieciowego.

Cały projekt zakładał komunikacje pomiędzy urządzeniami, zatem niezbędne było stworzenie aplikacji z wykorzystaniem programowania sieciowego. Najbardziej rozpowszechnionym jest

programowanie sieciowe z użyciem tak zwanych gniazd (socket'ów) [17]. Programowanie to opiera się właśnie o gniazda, które to są odpowiednikami deskryptorów plików w komunikacji przez sieć. Gniazdo identyfikuje konkretne połączenie sieciowe w aplikacji, jego obsługa jest analogiczna do obsługi plików, to znaczy można do gniazda pisać lub z gniazda odczytywać. Udostępnionych jest wiele funkcji wykonujących operacje wymagane przy komunikacji poprzez sieć. Podstawowe z nich to:

− funkcja socket,
− funkcja connect,
− funkcja bind,
− funkcja listen,
− funkcja accept,
− funkcja send,
− funkcja recv.

Jedne tych funkcji wykorzystywane będą przez aplikacje kliencką inne przez aplikację serwera. Podczas zestawiania połączenia zarówno po stornie klienta jak i serwera należy wykonać te funkcje w określonej kolejności. Dla klienta kolejność ta to *socket, connect, send/recv, close*, natomiast dla serwera *socket, bind, listen, accept, send/recv, close*. W celu zrozumienia, co się dzieje zarówno po stronie klienta jak i serwera dla uproszczenia można posłużyć się analogią z połączeniem telefonicznym. W przypadku aplikacji klienckiej, czyli w przypadku, gdy chcemy się do kogoś dodzwonić, *socket* jest ekwiwalentem posiadania telefonu, *connect* jest odpowiednikiem dzwonienia, które wymaga od nas znajomości numeru telefonu (adresu IP i numeru portu), funkcje *send/recv* są to odpowiednio mówienie/słuchanie naszego rozmówcy, natomiast *close* to zakończenie połączenia. W przypadku serwera, podobnie jak w przypadku klienta potrzebujemy telefonu – *socket*, następnie *bind*, który jest jakby powiedzeniem innym ludziom, że dany numer należy do nas, tak, aby mogli do nas zadzwonić, powiązanie numeru telefonu z konkretnym telefonem (funkcja ta wiąże adres IP oraz numer portu z danym gniazdem sieciowym). Funkcja *listen* to włączenie dzwonka, tak abyśmy mogli zwrócić uwagę, że ktoś do nas dzwoni, natomiast funkcja *accept* to odebranie połączenia (widzimy, kto do nas dzwoni i możemy to połączenie odebrać). Pozostałe funkcje można identyfikować podobnie jak w przypadku klienta. Szczegółowe informacje na temat programowania sieciowego można znaleźć w pozycji [17].

4.3. Wspólny format wymiany informacji.

Na potrzeby pracy stworzony został format wymiany informacji. Oparty on jest na wysyłaniu pomiędzy aplikacjami ustandaryzowanych wiadomości (*Message*). Wiadomość jest ciągiem bajtów kodowanych w ASCII. Tak, więc na przykład w przypadku potrzeby przesłania liczby 21, przesyłane są dwa bajty, będące kodami ASCII '2' oraz '1'. Format wiadomości musi być znany zarówno aplikacji serwera jak i aplikacji klienckiej. Pojedyncza wiadomość ma stałą ilość pół składających się z stałej ilości bajtów. Przyjęty format wiadomości przedstawiony został na rys.4.3.

```
const unsigned int PAYLOAD_SIZE = 1024;

struct RawMessage
{
    char msgId[4];
    char numOfMsgInFileTransfer[10];
    char bytesInPayload[5];
    char payload[PAYLOAD_SIZE];
};
```

Rys.4.3. Przyjęty format wiadomości.

Rozmiar wiadomości to 1043 bajty. Wiadomość składa się z czterech pól, pierwsze z nich to identyfikator wiadomości (*msgId*), określa, z jaka wiadomością mamy do czynienia. Wielkość tego pola to 4 bajty. W projekcie wyróżnione zostały następujące trzy główne typy wiadomości:

- Zapytania (*Request*), po otrzymaniu tego typu wiadomości należy odesłać odpowiedź (*Repsonse*) do aplikacji, która wysłała zapytanie. Natomiast aplikacja, która wysłała zapytanie czeka przez określony czas na odpowiedź, jeśli odpowiedź nie przyjdzie w określonym czasie uważane jest to za błąd.
- Odpowiedzi (*Response*), powinna przyjść od aplikacji, z której wcześniej zostało wysłane zapytanie, w innym przypadku jest uważane za błąd.
- Indykacje (*Indication*), wiadomości, które nie wymagają odpowiedzi ani żadnej reakcji ze strony odbierającej.

Przyjmuje się konwencję, że jeśli wiadomość posiada postfix _REQ jest zapytaniem, _RESP odpowiedzią oraz _IND indykacją. Na rys.4.4. przedstawiono wszystkie wiadomości używane w systemie. Dodatkowo prefix wiadomości to jest SERVER_ lub CLIENT_ oznacza kierunku wysyłania wiadomości, jeśli prefix to SERVER_ oznacza to ze jest to zapytania bądź indykacja wysyłana w kierunku od klienta do serwera (w przypadku odpowiedzi odwrotnie).

```
enum EMessageId
{
    CLIENT_WELCOME_MSG_IND,
    SERVER_TEST_FIRST_REQ,
    SERVER_TEST_FIRST_RESP,
    SERVER_TEST_SECOND_REQ,
    SERVER_TEST_SECOND_RESP,
    SERVER_SEND_FILE_REQ,
    SERVER_SEND_FILE_RESP,
    CLIENT_SEND_FILE_IND,
    SERVER_SEND_FILE_LIST_REQ,
    SERVER_SEND_FILE_LIST_RESP,
    SERVER_PARSE_DICOM_FILE_REQ,
    SERVER_PARSE_DICOM_FILE_RESP,
};
```

Rys.4.4. Zbiór wszystkich wiadomości używanych w systemie.

Następne pole jest liczbą wiadomości w procedurze przesyłania plików (*numOfMsgInFileTransfer*) o rozmiarze 10 bajtów. Informacja ta istotna jest przy wiadomości SERVER_SEND_FILE_RESP i mówi o tym ile wiadomości CLIENT_SEND_FILE_IND (zawierających części przesyłanego pliku) zostanie kolejno wysłanych. Procedura przesyłania plików zostanie omówiona w rozdziale 5.

Kolejne pole to ilość bajtów znaczących w informacji użytecznej (*bytesInPayload*). Informacja niesiona przez to pole przydatna jest podczas odczytywania informacji użytecznej po odebraniu wiadomości. Pole to ma rozmiar 5 bajtów. Maksymalny rozmiar informacji użytecznej to 1024 bajty, co można zapisać zgodnie z konwencją ASCII na 4 bajtach oraz jednym bajcie informującym o końcu pola.

Ostatnim polem jest informacja użyteczna(*payload*), czyli tak naprawdę właściwa informacja, którą trzeba przesłać. Przyjęto, że jednorazowo w wiadomości możemy przesłać 1024 bajty a więc jeden kilobajt informacji użytecznej.

Tak zdefiniowany format wiadomości pozwala na komunikację aplikacji bez względu na to, w jakich językach programowania zostały napisane. W przypadku systemu będącego tematem pracy aplikacja serwera napisana jest w języku C++ (jak podstawowa jednostka użyty jest typ char – 1 bajt) natomiast aplikacja kliencka w języki Java, (jako podstawowa jednostka użyty jest typ byte – typ char ma w języku Java 2 bajty).

5. Aplikacja serwera.

5.1. Zadania.

Do zadań aplikacji należą:

- Świadczenie usług bazodanowych – na serwerze przechowywany i udostępniany jest zbiór plików DCIOM
- Parsowanie plików w formacie DICOM – zadanie parsowania plików DICOM wykonywany jest przez serwer, jego wynik również zapisywany jest na serwerze.
- Wyodrębnianie z plików DICOM danych tekstowych – w wyniku parsowania powstają dwa pliki, jeden z nich, tekstowy zawiera zbiór wybranych danych pochodzących z elementów pliku, takich jak na przykład imię i nazwisko pacjenta, data badania, rodzaj badania
- Konwersja obrazu z pliku DIOCOM na format png – drugim plikiem powstałym w wyniku parsowania jest plik obrazu, wybrano format png ze względu na uniwersalność, lekkość oraz bezstratną kompresję tego formatu.
- Udostępnianie rezultatów parsowania aplikacjom klienckim – rezultaty parsowania są udostępnione klientom i mogą być w każdej chwili pobrane.
- Obsługa wielu klientów jednocześnie – dla każdego nowego połącznia pochodzącego z tego samego bądź różnych klientów tworzony jest osobny proces dziecko, dodatkowo proces rodzic cały czas oczekuje na nowe połączenia.

5.2. Środowisko pracy.

Aplikacja serwera napisana jest w języku C++ [18] w jego wersji oznaczonej numerem 14 to jest C++14 pochodzącej z grudnia 2014 roku [19]. Stacją bazową dla aplikacji serwera jest komputer klasy laptop z systemem operacyjnym Linux, idąc dalej jest to dystrybucja Fedora 21 [20]. Używanym kompilatorem jest g++ w wersji 4.9.2. Używanym edytorem jest gvim z kolorowaniem składni dla C++ oraz ctags w celu łatwego poruszania się po projekcie.

Do aplikacji zostały napisane testy na dwóch poziomach, poziomie jednostkowym (*Unit Tests*) oraz testy komponentowe (*Component Test*). Testy jednostkowe zostały napisane z wykorzystaniem frameworka testowego Google Test oraz Google Mock w wersji 1.7.0 [21].

W celu uzyskania łatwo testowalnych jednostek na poziomie pojedynczych klas, w projekcie zastosowano wzorzec projektowy wstrzykiwania zależności (*Dependency Injection*). Zastosowanie tego wzorca umożliwia zastępowanie wszystkich innych klas w obiekcie tak zwanymi mockami, czyli obiektami udającymi prawdziwe obiekty. Obiekt mocka przygotowany z pomocą framework'a Google Mock przedstawiony został na rys.5.1 Przykładowa klasa testowa testująca pojedyncza jednostkę, jaką jest klasa Dispatcher'a oraz pojedynczy test przedstawione zostały na rys.5.2.

```
#include "gmock/gmock.h"
#include "IUnixWrappers.hpp"
#include <iostream>

class UnixWrappersMock : public IUnixWrappers
{
public:
    MOCK_CONST_METHOD4(send, void(int, const Message*, size_t,int));
    MOCK_CONST_METHOD4(recv, ssize_t(int, Message*, size_t, int));
    MOCK_CONST_METHOD1(close, void(int));
    MOCK_CONST_METHOD0(fork, pid_t(void));
    MOCK_CONST_METHOD0(getPid, pid_t(void));
    MOCK_CONST_METHOD1(executeCommand, std::string(const char*));
};
```

Rys.5.1. Mock klasy UnixWrappers stworzony z wykorzystaniem Google Mock.

Testy modułowe napisano w języku C++. Stworzono proste środowisko uruchomieniowe w postaci prostej aplikacji konsolowej. Przykładowy test modułowy przedstawiono na rys.5.3. W każdym teście modułowym obecny jest nagłówek z krótkim opisem scenariusza testowego. Przedstawiony na rys.5.3. test modułowy testuje zachowanie serwera w przypadku żądania przesłania listy i przeslania wybranego z tej listy pojedynczego pliku tekstowego o znacznym rozmiarze. Przygotowane środowisko testowe umożliwia wykonanie wszystkich testów jednocześnie lub wykonanie wybranego pojedynczego testu.

Budowanie aplikacji serwera, testów jednostkowych oraz środowiska uruchomieniowego wraz z testami modułowymi wykonywane jest za pomocą programu make oraz przygotowanych plików typu Makefile [22]. Dla każdej z wymienionych grup stworzone są osobne Makefile, przez co możliwe jest zbudowanie samodzielnej aplikacji bez testów czy to jednostkowych czy to modułowych. Zawartość poszczególnych plików Makefile obecna jest w na płytach dołączonych do pracy i nie będzie w niniejszej pracy omawiana.

```cpp
#include "gtest/gtest.h"
#include "gmock/gmock.h"
#include "Dispatcher.hpp"
#include "ErrorHandlerMock.hpp"
#include "UnixWrapperMock.hpp"
#include "ServerSendFileRequestHandlerMock.hpp"
#include "ServerSendFileListRequestHandlerMock.hpp"
#include "ServerParseDicomFileRequestHandlerMock.hpp"
#include <string>

using ::testing::StrictMock;
using ::testing:: ;

class DispatcherTestSuite : public testing::Test
{
public:
    DispatcherTestSuite()
        : m_unixWrapperMock(std::make_shared<StrictMock<UnixWrappersMock>>()),
          m_serverSendFileRequestHandlerMock(std::make_shared<StrictMock<ServerSendFileRequestHandlerMock>>()),
          m_serverSendFileListRequestHandlerMock(std::make_shared<StrictMock<ServerSendFileListRequestHandlerMock>>()),
          m_serverParseDicomFileRequestHandlerMock(std::make_shared<StrictMock<ServerParseDicomFileRequestHandlerMock>>()),
          m_sut(m_unixWrapperMock,
                m_serverSendFileRequestHandlerMock,
                m_serverSendFileListRequestHandlerMock,
                m_serverParseDicomFileRequestHandlerMock) { }

    void fillMessageStructureForServerSendFileReq(Message& p_msg, const std::string& p_fileName);
    void checkCapturedStdOutput(const std::string& p_expectedText);

    std::shared_ptr<StrictMock<UnixWrappersMock>> m_unixWrapperMock;
    std::shared_ptr<StrictMock<ServerSendFileRequestHandlerMock>> m_serverSendFileRequestHandlerMock;
    std::shared_ptr<StrictMock<ServerSendFileListRequestHandlerMock>> m_serverSendFileListRequestHandlerMock;
    std::shared_ptr<StrictMock<ServerParseDicomFileRequestHandlerMock>> m_serverParseDicomFileRequestHandlerMock;
    Dispatcher m_sut;
};

TEST_F(DispatcherTestSuite, testIfServerSendFileListRequestHandlerWillBeCalledDuringDispatchingEventServerSendFileListReq)
{
    const int l_someSocket = 5;
    Message l_msg = {};
    l_msg.msgId = SERVER_SEND_FILE_LIST_REQ;
    strcpy(l_msg.payload, "File list request.");

    std::string l_expectedText = "PID: 0 | Case SERVER_SEND_FILE_LIST_REQ: received message - File list request.\n";
    testing::internal::CaptureStdout();

    EXPECT_CALL(*m_unixWrapperMock, getPid());
    EXPECT_CALL(*m_serverSendFileListRequestHandlerMock, handle(l_someSocket, l_msg));

    m_sut.dispatch(l_someSocket, l_msg);
    checkCapturedStdOutput(l_expectedText);
}
```

Rys.5.2. Klasa testowa DisaptcherTestSuite wraz z przykładowym testem jednostkowym

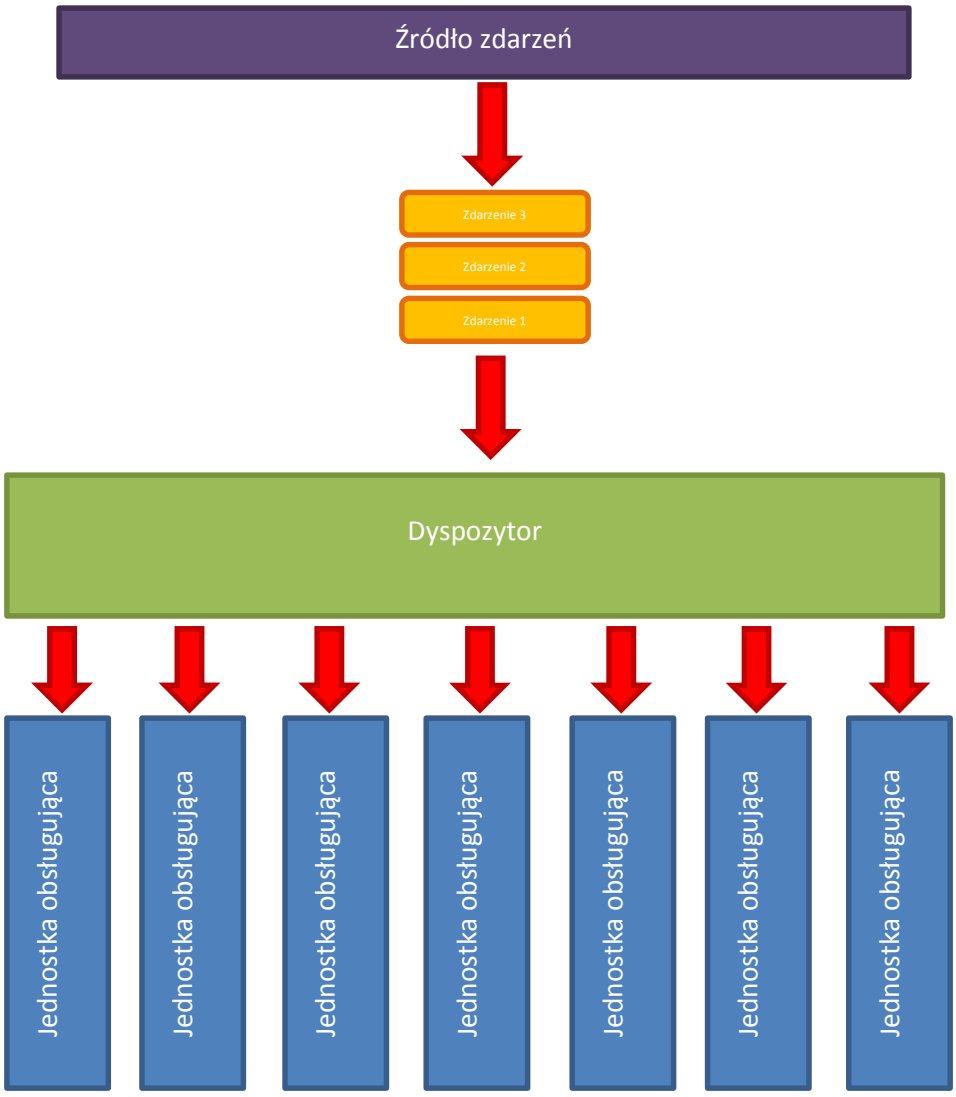
```
/*****************************************************************************
 * Test scenario:
 * Step1: Setup connection with server with address 192.168.254.1
 * Step2: Receive CLIENT_WELCOME_MSG_IND message from server
 * Step3: Send message SERVER_SEND_FILE_REQ to the server (large text file)
 * Step4: Receive SERVER_SEND_FILE_RESP message from server
 * Step5: Receive number of CLIENT_SEND_FILE_IND messages as defined in
 *        numOfMsgInFileTransfer field of SERVER_SEND_FILE_RESP
 * Step6: Check if received and requested file are equal
 * Step7: Close connection
*****************************************************************************/
void sendFileTransferRequestAndReceiveRequestedFileTest_largeTextFile(char** p_argv)
{
    std::cout << "Testcase " << __FUNCTION__ << " started." << std::endl;
//Step1
    initializeConnection(p_argv);
//Step2
    receiveMessageFromServer(CLIENT_WELCOME_MSG_IND);
//Step3
    std::string l_sourceFileName = "duzyTekstowy.txt";
    std::string l_sourceFilePath = "./moduleTest/plikiPrzykladowe/" + l_sourceFileName;
    Message l_sendline = {};
    l_sendline.msgId = SERVER_SEND_FILE_REQ;
    l_sendline.bytesInPayload = strlen(l_sourceFilePath.c_str()) + 1;
    strcpy(l_sendline.payload, l_sourceFilePath.c_str());

    g_unixWrapper.send(g_sockfd, &l_sendline);
//Step4
    receiveMessageFromServer(SERVER_SEND_FILE_RESP);
    int l_numberOfMessagesToCatch = g_receivedMessage.numOfMsgInFileTransfer;
    std::cout << "Number of messages to catch: " << l_numberOfMessagesToCatch << std::endl;
//Step5
    long long l_sumOfReceivedBytes = 0;
    std::string l_outFileName = "odebrany.txt";
    std::ofstream l_outFile(l_outFileName);
    for (int i = 0; i < l_numberOfMessagesToCatch; i++)
    {
        receiveMessageClientSendFileIndFromServer(l_outFile, l_sumOfReceivedBytes);
    }
    std::cout << "Amount of received bytes: " << l_sumOfReceivedBytes << std::endl;
    l_outFile.close();
//Step6
    checkIfRequestedAndReceivedFilesMatch("./plikiPrzykladowe/" + l_sourceFileName,
                                          l_outFileName);
//Step7
    g_unixWrapper.close(g_sockfd);
    std::cout << "Testcase " << __FUNCTION__ << " finished successfully." << std::endl;
}
```

Rys.5.3. Mock klasy UnixWrappers stworzony z wykorzystaniem Google Mock.

5.3. Programowanie sterowane zdarzeniami.

Przy pisaniu aplikacji serwera przyjęto metodologię programowania sterowanego zdarzeniami (*Event Driven Development*). [23] Metodyka ta polega na tym, że program wykonuję akcje w odpowiedzi na zdarzenia dotyczące programu. Zdarzenia te mogą być zewnętrzne (pochodzące np. od użytkownika czy z sieci) lub wewnętrzne (pochodzące z wnętrz programu). Metodyka ta jest przeciwnością programowania sterowanego przepływem sterowania. Metodyka programowania sterowanego zdarzeniami, zakłada, że istnieją następujące byty: zdarzenie (*Event*), kolejka zdarzeń (*Event Queue*), dyspozytor (*Disaptcher*), jednostka obsługująca (Handler). Byty te oraz zależności pomiędzy nimi przedstawione zostały na rys 5.4.



Rys.5.4. Byty stosowane w metodologii programowania sterowanego zdarzeniami.

Zdarzenie jest zdefiniowanym w programie bytem. Posiada swój identyfikator (*Id*) oraz ładunek użyteczny (*Payload*). W momencie przekazania zdarzenia do programu trafia ono na kolejkę zdarzeń. Zdarzenia zdejmowane są jedno po drugim z kolejki przez obiekt dyspozytora i w zależności od rodzaju zdarzenia, są one przekierowywane do odpowiednich jednostek obsługujących. Każda jednostka obsługująca obsługuje inny typ zdarzenia.

W systemie będącym tematem pracy zdarzeniem jest wiadomość *Message*, jak zostało przedstawione w rozdziale 4.3 posiada ona między innymi swój identyfikator oraz ładunek użyteczny. Wiadomość jest otrzymywana z gniazda, następnie dekodowana i przekazywana do dalej do dyspozytora. W aplikacji serwera istnieje obiekt dyspozytora – klasa *Disaptcher*, która po otrzymaniu wiadomości, wypakowuje jej identyfikator i szuka jednostki obsługującej dany typ wiadomości. Na rys 5.5 przedstawiono metodę *dispatch* klasy *Dispatcher* wykonującą wspomnianą czynność.

```
bool Dispatcher::dispatch(int p_clientSocket, const Message p_receivedMsg) const
{
    switch(p_receivedMsg.msgId)
    {
        case SERVER_TEST_FIRST_REQ:
        {
            std::cout << "PID: " << m_unixWrapper->getPid() << " | "
                      << "Case SERVER_TEST_FIRST_REQ: received message - "
                      << p_receivedMsg.payload
                      << std::endl;

            sendServerTestFirstResp(p_clientSocket);
            break;
        }
        case SERVER_TEST_SECOND_REQ:
        {
            std::cout << "PID: " << m_unixWrapper->getPid() << " | "
                      << "Case SERVER_TEST_SECOND_REQ: received message - "
                      << p_receivedMsg.payload
                      << std::endl;
            break;
        }
        case SERVER_SEND_FILE_LIST_REQ:
        {
            std::cout << "PID: " << m_unixWrapper->getPid() << " | "
                      << "Case SERVER_SEND_FILE_LIST_REQ: received message - "
                      << p_receivedMsg.payload
                      << std::endl;
            m_serverSendFileListRequestHandler->handle(p_clientSocket, p_receivedMsg);
            break;
        }
        case SERVER_SEND_FILE_REQ:
        {
            std::cout << "PID: " << m_unixWrapper->getPid() << " | "
                      << "Case SERVER_SEND_FILE_REQ: received message - "
                      << p_receivedMsg.payload
                      << std::endl;

            m_serverSendFileRequestHandler->handle(p_clientSocket, p_receivedMsg);
            break;
        }
        case SERVER_PARSE_DICOM_FILE_REQ:
        {
            std::cout << "PID: " << m_unixWrapper->getPid() << " | "
                      << "Case SERVER_PARSE_DICOM_FILE_REQ: received message - "
                      << p_receivedMsg.payload
                      << std::endl;

            m_serverParseDicomFileRequestHandler->handle(p_clientSocket, p_receivedMsg);
            break;
        }
        default:
        {
            std::cout << "PID: " << m_unixWrapper->getPid() << " | "
                      << "Unknown message identyfier" << std::endl;
            return false;
        }
    }
    return true;
}
        }
        default:
        {
            std::cout << "PID: " << m_unixWrapper->getPid() << " | "
                      << "Unknown message identyfier" << std::endl;
            return false;
        }
    }
    return true;
}
```

Rys.5.5. Metoda *dispatch* klasy *Dispatcher*.

W przypadku nieznalezienia jednostki obsługującej dla danego typu wiadomości wypisywany jest komunikat o błędnej wiadomości. W przypadku znalezienia jednostki obsługującej

obsługa wiadomości przekazywana jest do odpowiedniego obiektu jednostki obsługującej za pomocą metody *handle*.

5.4. Biblioteka dcmtk.

DCMTK jest biblioteką implementującą dużą cześć standardu DICOM. Biblioteka zawiera narzędzia do przetwarzania danych zapisanych w formacie DICOM. Całość napisana jest po części w języki ANSI C oraz C++. Projekt udostępniany jest na zasadach oprogramowania Open Source. Strona domowa biblioteki to http://dicom.offis.de/.

DCMTK składa się z następujących pakietów:

- config – pakiet zawierający narzędzia konfiguracyjne dla dcmtk,
- dcmdata – pakiet zajmujący się kodowaniem i dekodowaniem plików oraz aplikacje narzędziowe,
- dcmimage – pakiet wspierający obsługę barwnych obrazów DICOM,
- dcmimgle – pakiet zajmujący się przetwarzaniem obrazów,
- dcmjpeg – pakiet zawierający klasy do konwersji pomiędzy skompresowanymi JPEG i nieskompresowanymi reprezentacjami danych obiektu DICOM,
- dcmjpls − pakiet zawierający klasy do konwersji pomiędzy skompresowanymi JPEG-LS i nieskompresowanymi reprezentacjami danych obiektu DICOM,
- dcmnet – pakiet zajmujący się obsługą sieci,
- dcmpstat – pakiet implementujący API dla DICOM *Softcopy Grayscale Presentation State Storage*,
- dcmsign – pakiet związany z podpisami cyfrowymi,
- dcmsr – pakiet zawierający klasy do odczytu, zapisy, tworzenia, modyfikacji, dostępu do strukturyzowanych raportów DICOM,
- dcmtls – pakiet zawierający rozszerzenia pakietu dcmnet dla bezpieczeństwa,
- dcmwlm – pakiet zawierający prosty system archiwizacji danych,
- dcmqrdb – pakiet zawierający bazę danych dla danych DICOM,
- oflog – pakiet zawierający narzędzia logowania oparty o log4cplus,
- ofstd – pakiet zawierający klasy ogólnego przeznaczenia.

Z punktu widzenia projektu najważniejszymi pakietami okazały się pakiety config, dcmdata, dcmimgle. Pierwszy pakiet był niezbędny do podpięcia biblioteki DCMTK do projektu aplikacji serwera, zawiera informacje na temat instalacji biblioteki w systemie oraz

informacje na temat kolejności linkowania bibliotek statycznych poszczególnych pakietów w plikach Makefile.

Pakiet dcmdata umożliwił wczytanie samego pliku DICOM do programu, niezbędne było dołączenie pliku nagłówkowego dctk.h z tego pakietu. W pakiecie tym znajdują się definicja klasy *DcmFileFormat* oraz implementacja funkcji odczytującej plik – *loadFile()* wraz z definicjami odpowiednich flag sterujących takich jak na przykład *EGL_withoutGL* czy *DCM_MaxReadLength*. Użycie pakietu dcmdata jest niezbędne w każdym programie przetwarzającym dane DICOM z użyciem dcmtk.

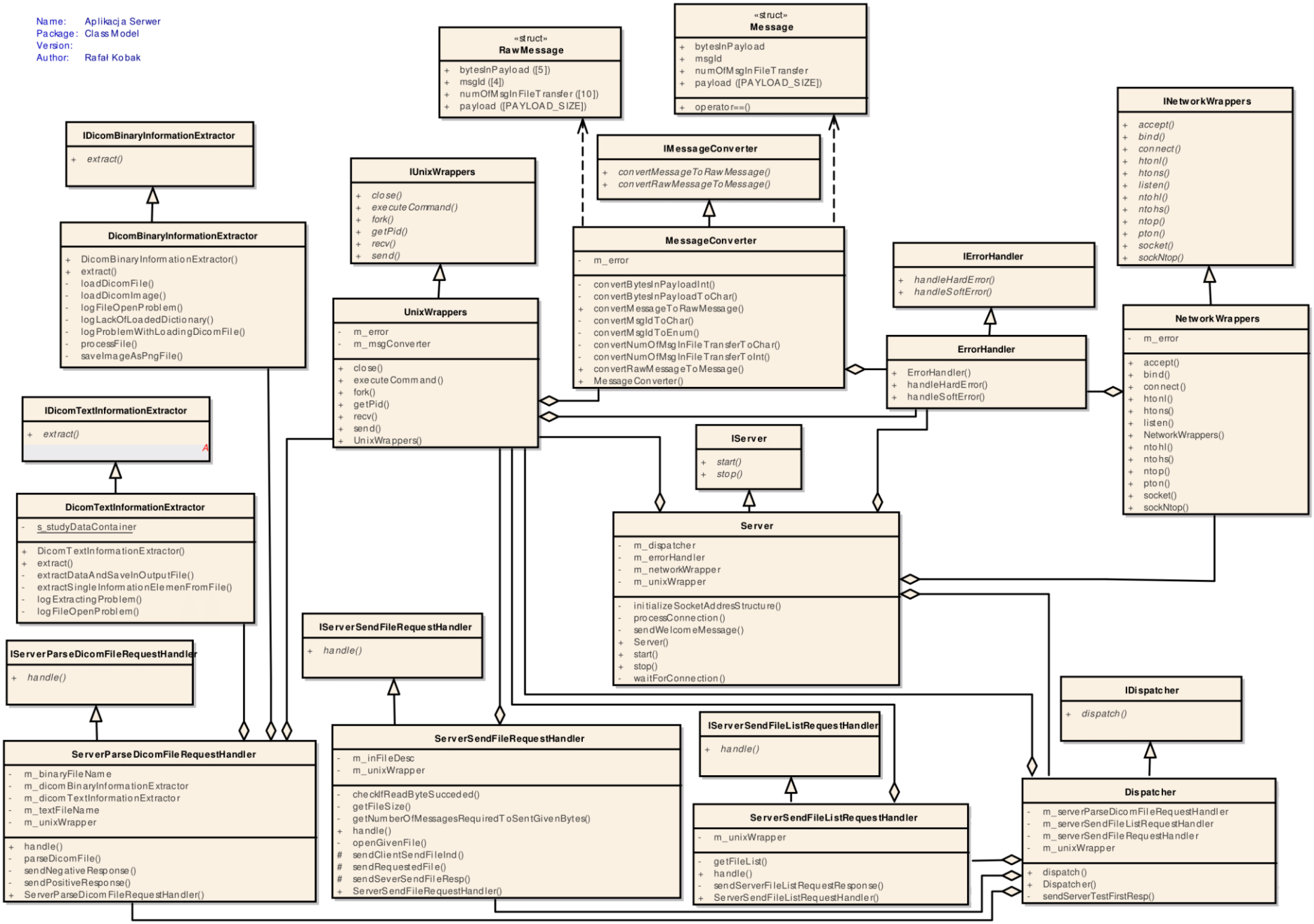
Pakiet dcmimgle umożliwił wyłuskanie z wczytanego za pomocą pakietu dcmdata pliku DICOM części związanej z obrazem. Z tego pakietu konieczny okazał się import plików nagłówkowych dcmimage.h oraz dipipng.h. W pliku dcmimage.h znajduje się definicja klasy obrazu DICOM – *DicomImage* oraz implementacja funkcji pomocniczych. W pliku dipipng.h pakietu dcmimgle znajduje się między innymi definicja klasy *DiPNGPlugin* zajmującej się konwersją *DicomImage* do formatu PNG.

5.5. Architektura aplikacji.

Architektura aplikacji przedstawiona została za pomocą diagramu klas na rys.5.6. Na diagramie tym przedstawiono hierarchię dziedziczenia, oraz zależności takie jak agregacje czy użycia pomiędzy poszczególnymi klasami. Każda z klas ze względu na zastosowanie wzorca wstrzykiwania zależności (dla celów testów jednostkowych) posiada interfejs będący w każdym przypadku klasą czysto abstrakcyjną składający się z deklaracji wszystkich publicznych metod z klas konkretnych. Przyjęto konwencję, że interfejsy mają identyczne nazwy jak klasy konkretne dziedziczące po nich z dodatkowym przedrostkiem I. I tak na przykład interfejsem klasy *Server* jest klasa *IServer*.

Dla uproszczenia i zachowania przejrzystości diagramu nie zaznaczono relacji pomiędzy klasą *Message* a pozostałymi klasami projektu (wyjątkiem jest klasa *MessageConverter* ze względu na swój szczególny charakter). Pozostałe klasy używają klasy *MessageConverter*, zatem na diagramie relacja powinna być oznaczona przerywaną linią zakończoną strzałką w kierunku klasy *Message*. Na diagramie linia ciągła zakończona białą strzałką oznacza relację dziedziczenia, linia przerywana zakończona strzałką oznacza relację użycia natomiast linia ciągła zakończona niewypełnionym rombem relację agregacji.

.



Rys.5.6. Diagram klas aplikacji serwera.

5.5.1. Klasa Server.

Główną klasą aplikacji jest klasa serwer. Za pomocą publicznej metody *start()* uruchamiana jest cała aplikacja. Start aplikacji serwera odbywający się w głównej funkcji programu *main()* przedstawiony został na rys5.7.

```
#include <iostream>
#include "Server.hpp"

int main()
{
    std::cout << "Hello World!" << std::endl;

    Server l_server;
    l_server.start();

    return 0;
}
```

Rys.5.7. Funkcja główna aplikacji serwera.

Klasa główna *Server* jak widać na diagramie klas z rys.5.6 oraz zawartości pliku nagłówkowego Server.hpp z rys.5.8 dziedziczy po *IServer* oraz agreguje w sobie obiekty klas *ErrorHandler*, *NetworkWrapper*, *UnixWrapper* oraz *Dispatcher*. Obiekty trzymane są w postaci inteligentnych wskaźników *std::unique_ptr* lub *std::shared_ptr*. Powodem przetrzymywania w klasie serwera wskaźników, a nie samych obiektów, jest oszczędność pamięci aplikacji. Jeżeli obiekt jest dalej współdzielony używany jest *std::shared_ptr*, w przeciwnym wypadku *std::unique_ptr*. Interfejsem klasy są metody *start()* oraz *stop()*.

```
#include "IServer.hpp"
#include "IErrorHandler.hpp"
#include "INetworkWrappers.hpp"
#include "IUnixWrappers.hpp"
#include "IDispatcher.hpp"

#include <memory>
#pragma once

class Server : public IServer
{
public:
    Server();

    void start() const override;
    void stop() const override;

private:
    SockAddrIn initializeSocketAddressStructure(const char* p_ipAddres,
                                                const unsigned int p_portNumber) const;

    void processConnection(int p_clientSocket) const;
    void waitForConnection(int p_serverSocket) const;
    void sendWelcomeMessage(int p_clientSocket) const;

    std::shared_ptr<IErrorHandler> m_errorHandler;
    std::unique_ptr<INetworkWrappers> m_networkWrapper;
    std::shared_ptr<IUnixWrappers> m_unixWrapper;
    std::unique_ptr<IDispatcher> m_dispatcher;
};
```

Rys.5.8. Plik nagłówkowy klasy Server.

Do utworzenia obiektu klasy *Server* nie potrzebujemy żadnych zależności. Podobnie metoda start nie przyjmuje żadnych argumentów. Metodę start można rozszerzyć o parametry określające numer portu oraz adres IP, na których ma nasłuchiwać aplikacja serwera. W obecnej implementacji informacje te są na stałe ustawione wewnątrz klasy. Najważniejsza metoda klasy Server, metoda *start()* przedstawiona została na rys.5.9.

```
void Server::start() const
{
    int l_serverSocket = m_networkWrapper->socket(AF_INET, SOCK_STREAM);
    SockAddrIn l_serverAddrStruct = initializeSocketAddressStructure("192.168.1.8", 9878);

    m_networkWrapper->bind(l_serverSocket,
                           reinterpret_cast<GenericSockAddr*>(&l_serverAddrStruct),
                           sizeof(l_serverAddrStruct));

    const unsigned int LISTENQ = 10;
    m_networkWrapper->listen(l_serverSocket, LISTENQ);

    waitForConnection(l_serverSocket);
}
```

Rys.5.9. Ciało metody *start()* klasy Server.

Pierwszą czynnością wykonywana w metodzie jest stworzenie tak zwanego gniazda sieciowego, na którym serwer będzie oczekiwał na nadchodzące połączenia. Stworzenie tego gniazda odbywa się za pomocą linuxowej funkcji *socket()* z pliku nagłówkowego sys/socket.h. Funkcja ta określa rodzaj protokołu komunikacyjnego.

Linuxowa funkcja *socket()* przyjmuje trzy parametry, natomiast widoczna na rys.5.9 jedynie dwa. Spowodowane jest to faktem, iż w projekcie używany jest specjalny obiekt opakowujący linuxowe funkcje do obsługi sieci. Klasą tego obiektu opakowującego jest *NetworkWrapper*. Zawiera ona metody opakowujące standardowe funkcje linuxowe z dodatkową obsługa błędów. Zaletą takiego rozwiązania jest poprawienie czytelności kodu, rozdzielenie poziomów abstrakcji (obsługa połączenia oraz obsługa błędów) oraz możliwość pisanie testów jednostkowych. W przypadku braku obiektu opakowującego możliwe byłoby stworzenie jedynie testów modułowych. Funkcja opakowująca *socket()* zostanie omówiona w podrozdziale opisującym klasę *NetworkWrappers*.

Wracając do linuxowej *socket()* przyjmuje ona trzy parametry. Są to kolejno rodzina protokołu *family,* rodzaj gniazda *type,* oraz protokół gniazda - *protocol*. Rodziną protokołu, na której nasłuchuje aplikacja serwera jest IPv4, dlatego jako pierwszy argument przekazywana jest stała AF_INET odpowiadająca tej rodzinie. W przypadku rodzaju gniazda, zdecydowano się na gniazdo strumieniowe toteż drugim argumentem wywołania funkcji jest SOCK_STREAM. Ostatni parametr to jest protokół gniazda ustawiany jest za pomocą parametru domyślnego na protokół TCP.

Kolejnym krokiem jest inicjalizacja tak zwanej struktury sieciowej *SockAddrInn.* Struktura ta zawiera adres IP oraz numer portu identyfikujące połączenie i jest niezbędna do działania funkcji sieciowych. Inicjalizacja struktury odbywa się za pomocą prywatnej metody *initializeSocketAddressStructure()* klasy Server. Metoda ta przyjmuje dwa argumenty, adres IP w postaci łańcucha znaków oraz numer portu w postaci liczby całkowitej. Metoda *initializeSocketAddressStructure()* dokonuje wymaganej konwersji przekazanych danych do formatu sieciowego oraz uzupełnia pozostałe pola struktury. Ciało tej metody przedstawione zostało na rys.5.10.

```
SockAddrIn Server::initializeSocketAddressStructure(const char* p_ipAddres,
                                                    const unsigned int p_portNumber) const
{
    SockAddrIn l_sockAddr = {};

    l_sockAddr.sin_family = AF_INET;
    l_sockAddr.sin_port = m_networkWrapper->htons(p_portNumber);
    m_networkWrapper->pton(AF_INET, p_ipAddres, &l_sockAddr.sin_addr);

    return l_sockAddr;
}
```

Rys.5.10. Ciało metody *initializeSocketAddressStructure()* klasy Server.

Następnie za pomocą funkcji *bind()* stworzone kilka linijek wcześniej gniazdo wiązane jest z strukturą sieciową. Funkcja ta przyjmuje typ gniazda, adres na strukturę sieciową, rzutowaną na specjalny surowy typ oraz jej rozmiar. Kolejnym krokiem jest ustawienie tak skonfigurowanego gniazda w stan nasłuchiwania. Operacja ta odbywa się za pomocą funkcji *listen().* Od tej pory możliwe jest połączenie się z serwerem. Funkcja *listen()* przyjmuje, jako argumenty wywołania przygotowane uprzednio gniazdo sieciowe oraz liczbą maksymalnych połączeń. Liczba maksymalnych połączeń mówi o tym ile maksymalnie przychodzących połączeń jest kolejkowanych do obsłużenia. W projekcie liczbę tą ustawiono na 10. Funkcje *bind()* oraz *listen()* są w projekcie opakowane podobnie jak funkcja *socket().*

Następnie przechodzimy do prywatnej metody *waitForConnection()*, w której jak opisuje to nazwa oczekujemy na połączenie. Metoda ta przyjmuje, jako argument wywołania gniazdo sieciowe w stanie nasłuchiwania. Ciało metody *waitForConnection()* przedstawione zostało na rys.5.11. Główną częścią tej metody jest nieskończona pętla while, w której oczekujemy na połączenie. Działanie programu zawieszone jest na funkcji *accept().* W przypadku przychodzącego połączenia, odczytywana jest struktura sieciowa *SockAddrIn* i zwracane jest nowe gniazdo utworzone dla tego konkretnego połączenia nowego połączenia. Następnie proces aplikacji dzielony jest na dwa procesy. Dla przychodzącego połączenia tworzony jest nowy proces dziecko, natomiast stary wciąż aktywny proces oczekuje na kolejne połączenia zamykając dla siebie uprzednio gniazdo utworzone poprzez funkcję *accept().*

```
void Server::waitForConnection(int p_serverSocket) const
{
    SockAddrIn l_clientAddrStruct = {};

    while(true)
    {
        std::cout << "PID: " << m_unixWrapper->getPid() << " | "<< "Waiting for connection..." << std::endl;

        socklen t l clientLen = sizeof(l clientAddrStruct);
        int l_clientSocket = m_networkWrapper->accept(p_serverSocket,
                                                      reinterpret cast<GenericSockAddr*>(&l clientAddrStruct),
                                                      &l clientLen);
        if (m_unixWrapper->fork() == 0)
        {
            m_unixWrapper->close(p_serverSocket);

            std::cout << "PID: " << m_unixWrapper->getPid() << " | Connection from: "
                      << m_networkWrapper->sockNtop(reinterpret_cast<GenericSockAddr*>(&l_clientAddrStruct))
                      << std::endl;

            processConnection(l_clientSocket);
            exit(0);
        }
        m_unixWrapper->close(l_clientSocket);
    }
}


void Server::processConnection(int p_clientSocket) const
{
    const unsigned int MAXLINE = 4096;
    ssize_t l_receivedBytes = 0;
    Message l_receivedMessage = {};
    bool l_status = true;

    sendWelcomeMessage(p_clientSocket);

again:
    while ((l_receivedBytes = m_unixWrapper->recv(p_clientSocket, &l_receivedMessage)) > 0)
    {
        l_status = m_dispatcher->dispatch(p_clientSocket, l_receivedMessage);
        if (l status == false)
        {
            m_errorHandler->handleHardError("processConnection: dispatching error");
        }
    }

    if (l receivedBytes < 0 && errno == EINTR)
    {
        goto again;
    }
    else if (l receivedBytes < 0)
    {
        m_errorHandler->handleHardError("processConnection: recv error");
    }
}
```

Rys.5.12. Ciała metod *waitForConnection()* oraz *processConnection()* klasy Server.

Nowy proces dziecko dla nadchodzącego połączenia najpierw zamyka dla siebie gniazdo nasłuchujące, następnie wysyła na standardowe wyjście informacje o źródle połączenia i przechodzi do prywatnej metody *processConnection()* przyjmującej, jako argument wywołania gniazdo nowego połączenia. Ciało metody *processCnnection()* przedstawione zostało na rys.5.12.

Metoda *processConnection()* zajmuje się obsługa odebranego połączenia. W pierwszej kolejności do klienta, który nawiązał połączenie wysyłana jest wiadomość powitalna – metoda *sendWelcomeMessage()*. Ciało tej metody przedstawione zostało na rys.5.13. Wiadomość ta budowana jest w oparciu o strukturę opisaną w rozdziale 4.3. W przypadku wiadomości powitalnej identyfikatorem wiadomości jest CLIENT_WELCOME_MSG_ID. W ładunku użytecznym zawarta jest krótka wiadomość powitalna w postaci łańcucha znaków. Przygotowana struktura *Message* jest oddelegowywana do obiektu klasy *UnixWrappers* który zajmuje się między innymi wysyłaniem wiadomości na wskazane gniazdo.

```
void Server::sendWelcomeMessage(int p_clientSocket) const
{
    const std::string l_welcomeMessageContent = "Welcome on server!";
    Message l_welcomeMessage = {};
    l_welcomeMessage.msgId = CLIENT_WELCOME_MSG_IND;
    l_welcomeMessage.bytesInPayload = strlen(l_welcomeMessageContent.c_str()) + 1;
    strcpy(l_welcomeMessage.payload, l_welcomeMessageContent.c_str());

    m_unixWrapper->send(p_clientSocket, &l_welcomeMessage);
}
```

Rys.5.13. Ciało metody *sendWelcomeMessage()* klasy Server.

Wiadomość zostaje wysłana i sterowanie zostaje zwrócone do metody *processConnection()*. Dalej w wyżej wymienionej metodzie odczytywana jest w pętli while wiadomość z gniazda połączenia klienta, każdorazowo odczytywane jest 4096 bajtów, czyli rozmiar struktury *Message*. W przypadku niepowodzenia albo podejmowana jest próba ponownego odczytu wiadomości z gniazda albo wypisywany jest komunikat o błędzie i kończone jest działanie procesu.

W przypadku udanego odczytu wiadomości, odczytana wiadomość jest oddelegowywana do dyspozytora reprezentowanego w projekcie przez klasę *Dispatcher* za pomocą metody *dispatch()*. Wynika przetwarzania zwracany jest przez obiekt dyspozytora i w zależności od rezultatu proces kończy się wypisaniem komunikatu o błędzie, bądź kończy się sukcesem i kończony jest za pomocą *exit(0)* w funkcji *waitForConnection()*.

5.5.2. Klasa Dispatcher.

Klasa dyspozytora *Dispatcher* decyduje o tym gdzie dana wiadomość *Message* zostanie oddelegowana do przetwarzania. Klasa ta podobnie jak klasa serwera posiada interfejs w postaci klasy czysto abstrakcyjnej z jedną metodą *dispatch()* - rys.5.6. Podobnie na rys5.6. oraz w zawartości pliku nagłówkowego rys5.14 klasy widoczne jest, że klasa ta agreguje trzy obiekty jednostek obsługujących *handlerów ServerSendFileListRequestHandler, ServerSendFileRequestHandler, ServerParseDicomFileRequestHandler* oraz obiekt klasy *UnixWrappers*. Wszystkie te obiekty składowe przekazywane są do obiektu dyspozytora za pomocą konstruktora. Sam obiekt dyspozytora agregowany jest jedynie przez klasę *Server*.

Jedyną metodą publiczną jest metoda *dispatch()* przyjmująca, jako argumenty wywołania gniazdo sieciowe połączenia oraz otrzymaną wiadomość. Ciało metody *dispatch()* przedstawione zostało na rys5.5. Metoda ta sprawdza identyfikator otrzymanej wiadomości *msgId* i jeżeli identyfikator ten zostaje rozpoznany, na standardowe wyjście wypisywana jest stosowna informacja oraz działanie oddelegowywane jest do obiektu konkretnej jednostki obsługującej za pomocą metody *handle()* danej jednostki. Po przetworzeniu wiadomości przez jednostkę obsługującą zwracana jest wartość *true* informująca obiekt klasy *Server* o tym, iż przetwarzanie wiadomości powiodło się.

W przypadku, gdy dla identyfikatora otrzymanej wiadomości nie jest zdefiniowana żadna jednostka obsługująca, wyświetlany jest stosowny komunikat oraz zwracana jest wartość *false* do obiektu klasy Server.

Pierwsze dwa bloki w metodzie dla wiadomości o identyfikatorach SERVER_TEST_FIRST_REQ oraz SERVER_TEST_SECOND_REQ są stworzone na potrzeby testów komponentowych i nie posiadają własnych jednostek obsługujących. Podobnie prywatna *metoda sendServerTestFirstResp()* stworzona jest na potrzeby testów komponentowych.

```cpp
#include "IDispatcher.hpp"
#include "IErrorHandler.hpp"
#include "IUnixWrappers.hpp"
#include "IServerSendFileRequestHandler.hpp"
#include "IServerSendFileListRequestHandler.hpp"
#include "IServerParseDicomFileRequestHandler.hpp"
#include "CommonTypes.h"

#include <memory>

#pragma once

class Dispatcher : public IDispatcher
{
public:
    Dispatcher(std::shared_ptr<IUnixWrappers> p_unixWrapper,
               std::shared_ptr<IServerSendFileRequestHandler> p_serverSendFileRequestHandler,
               std::shared_ptr<IServerSendFileListRequestHandler> p_serverSendFileListRequestHandler,
               std::shared_ptr<IServerParseDicomFileRequestHandler> p_serverParseDicomFileRequestHandler);

    bool dispatch(int p_clientSocket, const Message p_receivedMsg) const override;

private:
    void sendServerTestFirstResp(int p_clientSocket) const;

    std::shared_ptr<IUnixWrappers> m_unixWrapper;
    std::shared_ptr<IServerSendFileRequestHandler> m_serverSendFileRequestHandler;
    std::shared_ptr<IServerSendFileListRequestHandler> m_serverSendFileListRequestHandler;
    std::shared_ptr<IServerParseDicomFileRequestHandler> m_serverParseDicomFileRequestHandler;
};
```

Rys.5.14. Zawartość pliku nagłówkowego Dispatcher.hpp.

5.5.3. Klasa ServerSendFileListRequestHandler.

Jest to klasa jednostki obsługującej *Handler* dla wiadomości o identyfikatorze SERVER_SEND_FILE_LIST_REQ. Sama klasa dziedziczy po klasie interfejsowej udostępniającej jak wszystkie interfejsy klas jednostek obsługujących metodę *handle*(). Klasa *ServerSendFileListRequestHandler* agreguje w sobie obiekt klasy *UnixWrappers*, który jest opakowaniem dla funkcji linuxowych. Sam obiekt omawianej klasy zawierany jest przez klasę *Dispatcher*. Plik nagłówkowy klasy przedstawiony został na rys.5.15.

```
#include "IServerSendFileListRequestHandler.hpp"
#include "IUnixWrappers.hpp"
#include "CommonTypes.h"
#include <memory>
#include <string>

#pragma once

class ServerSendFileListRequestHandler
 : public IServerSendFileListRequestHandler
{
public:
    ServerSendFileListRequestHandler(std::shared_ptr<IUnixWrappers> p_unixWrapper);
    void handle(int p_clientSocket,
                const Message& p_receivedMsg) const override;

private:
    void sendServerFileListRequestResponse(int p_clientSocket,
                                           std::string p_fileList) const;
    std::string getFileList() const;
    std::shared_ptr<IUnixWrappers> m_unixWrapper;
};
```

Rys.5.15. Zawartość pliku nagłówkowego klasy *ServerSendFileListRequestHandler*.

Zadaniem omawianej klasy jest pobranie listy plików dostępnych na serwerze realizowane za pomocą prywatnej metody *getFileList()* i odesłanie jej do klienta, który przysłał żądanie w postaci wiadomości SERVER_SEND_FILE_LIST_RESP.

Na rys.5.16. przedstawione zostało ciało metody publicznej *handle()*. W pierwszej kolejności pobierana jest lista plików – getFileList(), następnie sprawdzane jest czy pobrana lista jest poprawna. W przypadku, gdy lista jest niepoprawna, lista ustawiana jest na pusty łańcuch znaków. Kolejną czynnością jest wysłanie odpowiedzi za pomocą metody *sendServerFileListRequestResponse()*.

```
void ServerSendFileListRequestHandler::handle(int p_clientSocket,
                                              const Message& p_receivedMsg) const
{
    std::string l_fileList = getFileList();
    if (l_fileList == "ERROR")
    {
        l_fileList = "";
    }
    sendServerFileListRequestResponse(p_clientSocket, l_fileList);
}
```

Rys.5.16. Ciało metody publicznej handle() klasy *ServerSendFileListRequestHandler*.

Ciało metody *getFileList()* przedstawione zostało na rys.5.17. W metodzie tej w pierwszej kolejności pobierana jest bezwzględna ścieżka do repozytorium git, w którym znajduje się aplikacja serwera. Ścieżka pobierana jest za pomocą polecenia *git rev-parse --show-toplevel* natomiast sama komenda jest wykonywana w wierszu poleceń za pomocą metody pu blicznej *executeCommand()* obiektu klasy *UnixWrappers*.

```
std::string ServerSendFileListRequestHandler::getFileList() const
{
    std::string l_repositoryRootPath
         = m_unixWrapper->executeCommand("git rev-parse --show-toplevel");

    if(l_repositoryRootPath == "ERROR")
    {
        return l_repositoryRootPath;
    }

    l_repositoryRootPath.back() = '/';
    std::string l_filesPath = l_repositoryRootPath +
                              "projekt/moduleTest/plikiTestyAndroid/";

    return m_unixWrapper->executeCommand(("ls " + l_filesPath).c_str());
}
```

Rys.5.17. Ciało metody prywatnej *getFileList()* klasy *ServerSendFileListRequestHandler*.

W zmiennej l_repositoryRootPath przechowywany jest wynik wykonania polecenia a więc wspomniana ścieżka. Dalej sprawdzana jest jej poprawność, następnie poprzez konkatenacje dodawana jest dalsza część ścieżki już do docelowej lokalizacji, w której znajdują się pliki DICOM udostępniane przez serwer. Tak przygotowana pełna ścieżka jest sprawdzana za pomocą polecenie linuxowego *ls*, ponownie wykonanego w konsoli za pomocą metody *executeCommand()*. Rezultat ostatniej komendy zwracany jest przez metodę *getFileList()* do metody *handle()* w postaci obiektu *std::string*.

```
void ServerSendFileListRequestHandler
          ::sendServerFileListRequestResponse(int p_clientSocket,
                                              std::string p_fileList) const
{
    Message l_msg = {};
    l_msg.msgId = SERVER_SEND_FILE_LIST_RESP;
    l_msg.bytesInPayload = strlen(p_fileList.c_str());
    strcpy(l_msg.payload, p_fileList.c_str());

    m_unixWrapper->send(p_clientSocket, &l_msg);
}
```

Rys.5.18. Ciało metody prywatnej *sendFileListRequestResponse()*.

Ciało metody *sendFileListRequestResponse()* przedstawione zostalo na rys.5.18. Zadaniem tej metody jest odesłanie odpowiedzi z listą plików. W pierwszej kolejności uzupełniana jest wiadomość, jej identyfikator SERVER_SEND_FILE_LIST_RESP, dalej ładunek użyteczny to jest uzyskana wcześniej lista plików przekazana do metody w postaci parametru, oraz wielkość ładunku użytecznego. Uzupełniona wiadomość wysyłana jest do klienta za pomocą metody publicznej *send()* klasy *UnixWrappers*.

5.5.4. Klasa ServerSendFileRequestHandler.

//opis ważniejszych klas wszystkie Handlery, konwerter wiadomości wzmianka o wrapperach)

5.6. Możliwość rozwoju oraz performance.

//dodawanie dodatkowych handlerów o perfomrance, wyniki z callgrinda oraz czegoś do pamieci memcheck

6. Aplikacja kliencka.

6.1. Zadania.

Aplikacji klienta ma być z założenia lekka i być zdolna do działania na urządzeniach mobilnych. Do zadań aplikacji klienckiej należą:

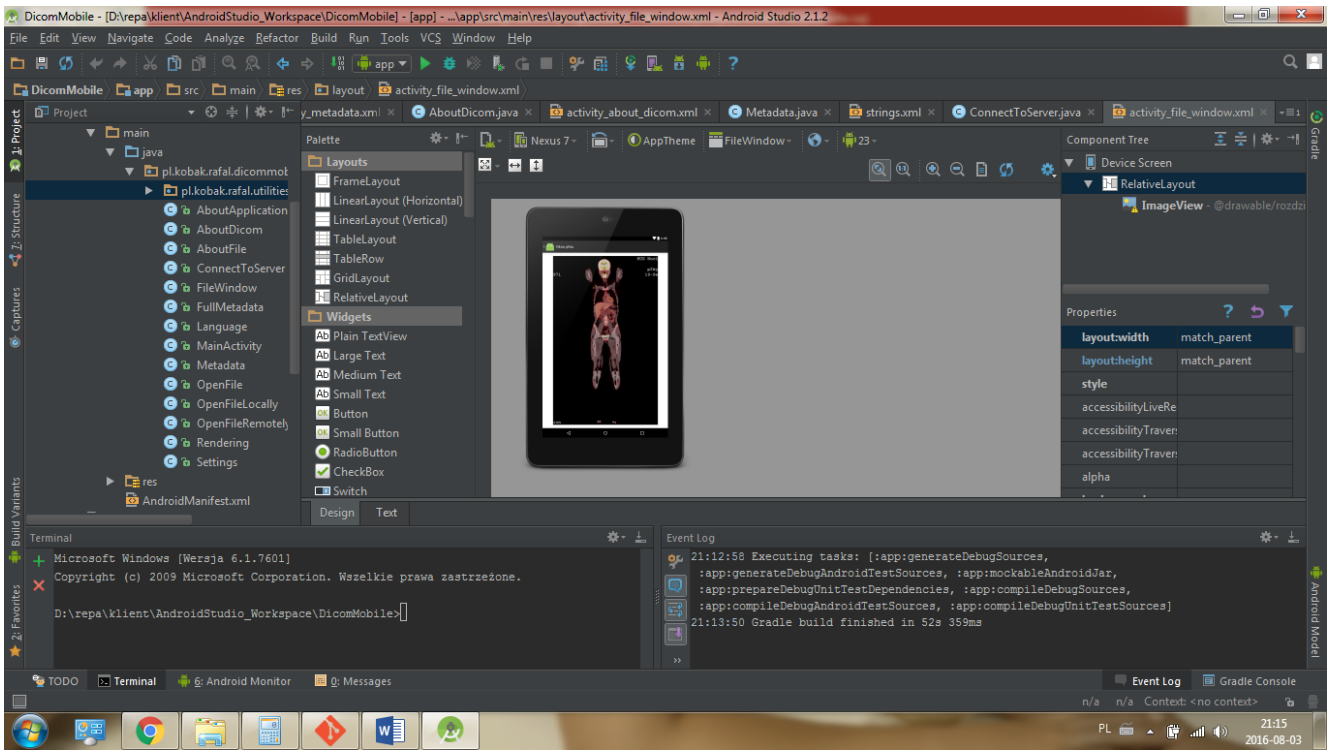
- Możliwość połączenia z aplikacją serwera poprzez protokół TCP.
- Prezentacja wyników parsowania aplikacji serwera w postaci danych graficznych oraz danych tekstowych.
- Działanie na urządzeniach mobilnych typu tablet oraz smartfon.

6.2. Środowisko programistyczne.

Aplikacja kliencka ma w założeniu działać po kontrolą systemu android. Jako urządzenie testowe wybrano tablet z systemem android w wersji 4.4.4 − KitKat. Urządzeniem testowym jest tablet Dell Venue 7 3740. Tablet ten posiada procesor dwurdzeniowy Intel Atom Z3460 o częstotliwości taktowania 1,6 GHz z 1 MB pamięci cache. Pamięć operacyjna RAM wynosi 1024 MB DDR3. Tablet posiada pamięć wewnętrzną flash o pojemności 16 GB. Przekątna ekranu to 7 cali.

Aplikacja kliencka napisana jest w języku Java w wersji 8 [24]. Język Java w wersji pod aplikacje na platformę Andorid różni się pod pewnymi względami od konwencjonalnej Java'y. Różnice te oraz specyfika programowania w systemie Android jakże odmienna od standardowego programowania w języku Java opisane zostały w [25] oraz [26]. Tworzenie aplikacji dzieli się na dwie główne części, część związaną z interfejsem użytkownika (*frontend*), regulowaną w najwygodniejszy sposób za pomocą plików XML oraz część związaną algorytmem działania aplikacji, obsługa zdarzeń, obsługa sieci (*backend*) regulowaną za pomocą plików z rozszerzeniem *java*.

Całe środowisko programistyczne wraz z edytorem, kompilatorem, środowiskiem budowania oraz środowiskiem uruchomieniowym dostarczone jest przez firmę Google w postaci oprogramowania o nazwie Android Studio. W takcie pisania aplikacji klienckiej korzystano z wersji 2.1.2 aplikacji. Okno aplikacji przedstawiono na rys.6.1. Wraz z środowiskiem firma Google udostępnia szczegółową dokumentację do wersji języka Java przystosowanej do systemu android [27].



Rys.6.1. Okno aplikacji Android Studio.

Głównym plikiem konfiguracyjnym aplikacji jest plik *AndoridManifest.xml*. W pliku tym zdefiniowane są podstawowe ustawienia aplikacji takie jak jej nazwa, uprawnienia w systemie Android. Dodatkowo plik ten sprzęga wszystkie pozostałe elementy aplikacji takie jak aktywności oraz zasoby. Aplikacja składa się z kolejnych aktywności uporządkowanych w hierarchie. Pliki aktywności są plikami XML, wyjściową aktywnością będąca aktywnością główną jest *activity_main.xml*. Z aktywności tej możliwe jest uruchamianie kolejnych aktywności i w ten sposób tworzona jest hierarchia ekranów. Pliki aktywności powinny być umieszczone w katalogu *layout* projektu. Na poszczególnych aktywnościach można tworzyć oraz oprogramować kolejne elementy takie jak przyciski, pola tekstowe, check box'y. Dalej w poszczególnych aktywnościach można prezentować zasoby aplikacji takie jak elementy graficzne, elementy tekstowe. W przypadku tworzenia aplikacji na urządzenia mobilne istotne jest stworzenie kilku wersji zasobów graficznych, jeżeli aplikacja ma być przeznaczona na urządzenia o różnej przekątnej ekranu. Zasoby graficzne aplikacji powinny być umieszczone w folderze *drawable* projektu. Szczególnym rodzajem zasoby graficznego jest ikona aplikacji. Podobnie jak w przypadku zwyczajnych zasobów graficznych, ikona aplikacji powinna być przygotowana w kilku rozmiarach. Ikona powinna być umieszczona w folderze *mipmap*. Zasoby tekstowe mogą występować w postaci plików tekstowych oraz ciągów znaków. Ciągami znaków są na przykład tytuły ekranów, napisy na przyciskach. Przyjmuje się, że wszystkie teksty w aplikacji powinny być przechowywane w specjalnym zasobie *strings.xml*. Umieszczanie wszystkich zasobów tekstowych w tym pliku umożliwia łatwe tworzenie

różnych wersji językowych aplikacji. Innymi zasobami aplikacji są pliki dimens.xml, w których przechowywane są wymiary używane w aplikacji, na przykład wymiary przycisków, pól tekstowych oraz styles.xml, w których przechowywane są style poszczególnych elementów.

Jeszcze jednym plikiem, o którym warto wspomnieć jest plik *build.gradle.* W pliku tym definiujemy wersję aplikacji. Po każdej zmianie aplikacji możemy nadać numer kodowy wersji.

6.3. Architektura aplikacji.

//hierarchia ekranów, opis ekranu głównego, menu, opis ekranu do połączenia z serwerem, opis ekranu do wyświetlania rezultatu parsowania, opis ekranu z wersjami językowymi, opis ekranów opisowych o DICOM o aplikacji + zrzuty dla każdego ekranu (znaleźć apke na andorida robiąca zrzuty ekranu)

6.4. Możliwość rozwoju.

//dodatkowe ekrany, system Teledicom, połączeni Bluetooth

7. Historie użytkowników.

//tutaj chciałbym opisać przebieg zachowania całego systemu (od strony aplikacji serwera oraz aplikacji klienta) przy podłączaniu do serwera, żądaniu przesłania pojedynczego pliku, żądania parsowania pliku DICOM. Wyświetlanie samego obrazu z pliku DICOM, prezentacja danych tekstowych z pliku DIOCM.

8. Podsumowanie.

,/*co się udało z założeń zrealizować co nie, możliwość zastosowania, napotkane problemu i rozwiązania, możliwość rozwoju*/

- na płycie kod całości aplikacji

9. Bibliografia.

[1] Oleg S. Pianykh, *Digital Imaging and Communications in Medicine (DICOM)*, Springer Boston 2008.

[2] NEMA, *DICOM PS3.1 2015c - Introduction and Overview*, 2015

[3] NEMA, *DICOM PS3.3 2015c - Information Object Definitions*, 2015

[4] NEMA, *DICOM PS3.4 2015c - Service Class Specifications*, 2015

[5] NEMA, *DICOM PS3.5 2015c - Data Structures and Encoding Standard DICOM*, 2015

[6] NEMA, *STRATEGIC DOCUMENT*, 2014

[7] Beata Brzozowska, Wykłady z przedmiotu *Praktyka z diagnostycznych metod nieradiacyjnych DICOM,* Wydział Fizyki Uniwersytetu Warszawskiego 2013

[8] Strona domowa standardu DICOM
http://dicom.nema.org

[9] Strategie rozwoju standard DICOM
http://medical.nema.org/dicom/geninfo/strategy.pdf

[10] Broszura na temat DICOM ze strony towarzystwa NEMA
http://medical.nema.org/dicom/geninfo/Brochure.pdf

[11] Strona internetowa Mirosława Sochy na temat standardu DICOM
http://home.agh.edu/~socha/pmwiki/pmwiki.php/DICOM/

[12] Strona główna biblioteki DCMTK
http://dicom.offis.de/dcmtk

[13] Strona główna biblioteki gdcm
http://gdcm.sourceforge.net/wiki/

[14] Strona domowa projektu aplikacji medycznej TeleDICOM
http://www.teledicom.pl/index.php/pl/

[15] Strona anglojęzycznej Wikipedii na temat systemów PACS
https://en.wikipedia.org/wiki/Picture_archiving_and_communication_system

[16] System PACS Eskulap
https://www.systemeskulap.pl/oferta/pacs/

[17] UNIX Network Programming – the sockets networking API Third Edition, volume 1 and volume 2, W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, Addison Wesley 2003.

[18] Język C++ Szkoła programowania wydanie VI, Stephen Prata, Helion Gliwice 2012

[19] Portal na temat języka C++ wraz z standardem
http://www.cplusplus.com/

[20] Projekt Fedora, strona domowa https://getfedora.org/pl/

[21] Framework testowy Google Test oraz Google Mock, strona domowa projektu, https://github.com/google/googletest

[22] GNU Make, Richard M. Stallman, Roland McGranth, Paul D. Smith, Wrzesień 2011.

[23] Event-Driven Programming: Introdution, Tutorial, History. Stephen Ferg 2006.

[24] Java podstawy wydanie IX, Cay S. Horstman, Gary Cornell, Helion 2014.

[25] Android Studio – podstawy tworzenia aplikacji, Andrzej Stasiewicz, Helion 2015.

[26] Platforma Android – nowe wyzwania, Erik Hellman, Helion 2014.

[27] Portal na temat programowania w systemie android z wykorzystaniem środowiska Andorid Studio, https://developer.android.com/index.html