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, 2016

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. Przy rozdzielczościach oraz mocach obliczeniowych dzisiejszych tabletów system taki w sposób wygodny umożliwiłby lekarzowi dostęp do danych medycznych pacjenta bezpośrednio przy jego łóżku. W jednym miejscu zawarte byłyby takie informacje jak wyniki badań w postaci zdjęć czy danych tekstowych. System sprawdzałby się szczególnie w przypadku pacjentów leżących, kiedy istnieje potrzeba obrazowego wyjaśnienia pacjentowi przyszłego zabiegu czy badania.

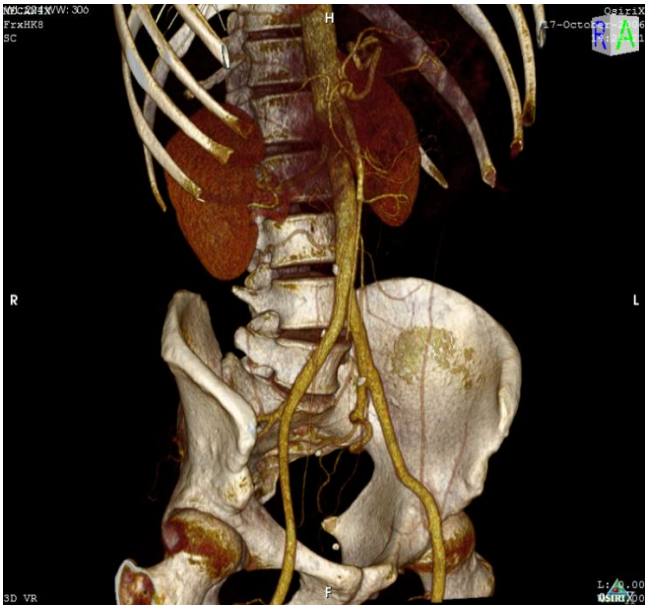
Dodatkowo system byłby użyteczny przy przygotowywaniu się lekarza do zabiegu, kiedy wszystkie potrzebne informacje zgromadzone są w jednym miejscu, dostępne pod ręką. Co więcej istnieje możliwość integracji systemu będącego tematem niemniejszej pracy z systemem TeleDICOM, co dodatkowo rozszerzyłoby jego funkcjonalność o możliwość szybkiej zdalnej konsultacji wyników badań z lekarzami w innych placówkach.

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 mogą 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.

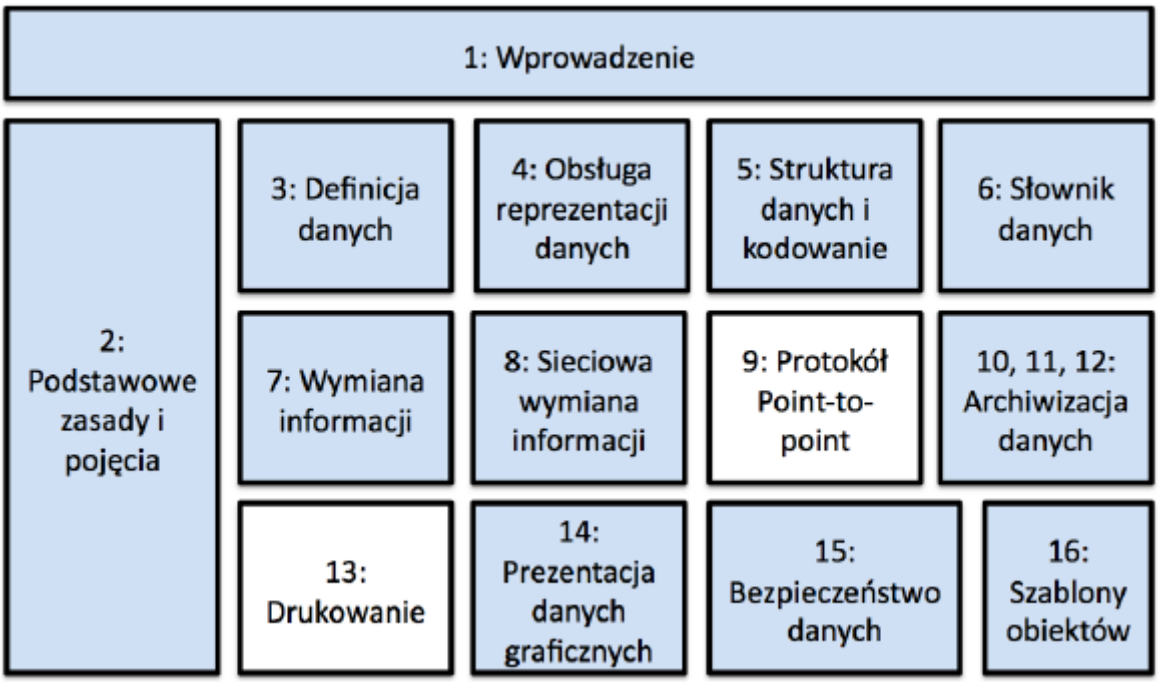
W roku 1993 ukazała się trzecia wersja standardu ACR-NEMA. Nazwa standardu została zmieniona na DICOM (Digital Imaging and Communications in Medicine).

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.

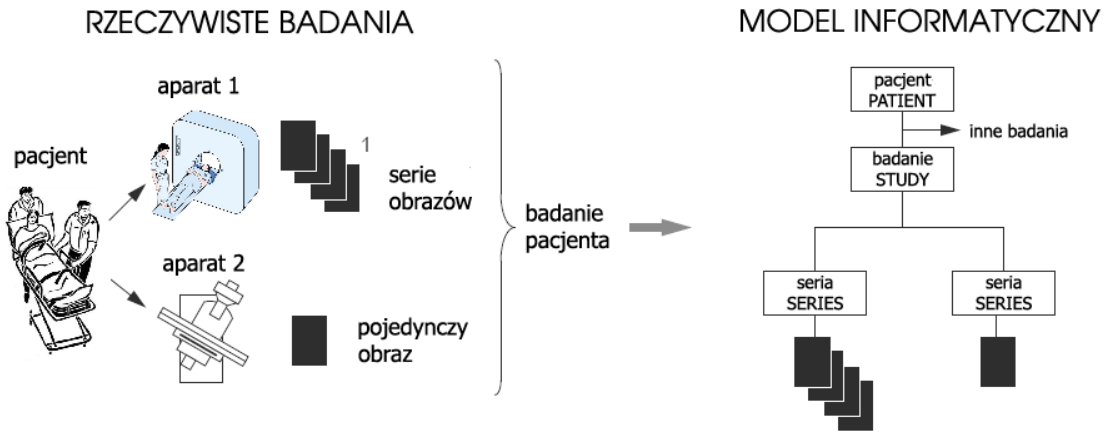
Rozdział trzeci standardu opisuje sposób definicji danych a więc określa między innymi ilość klas obiektów danych w skrócie IOC (Information Object Classes), które to zapewniają abstrakcyjne odzwierciedlenie rzeczywistych jednostek związanych z obrazowaniem medycznym takich jak dawka promieniowania, 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 przez 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.

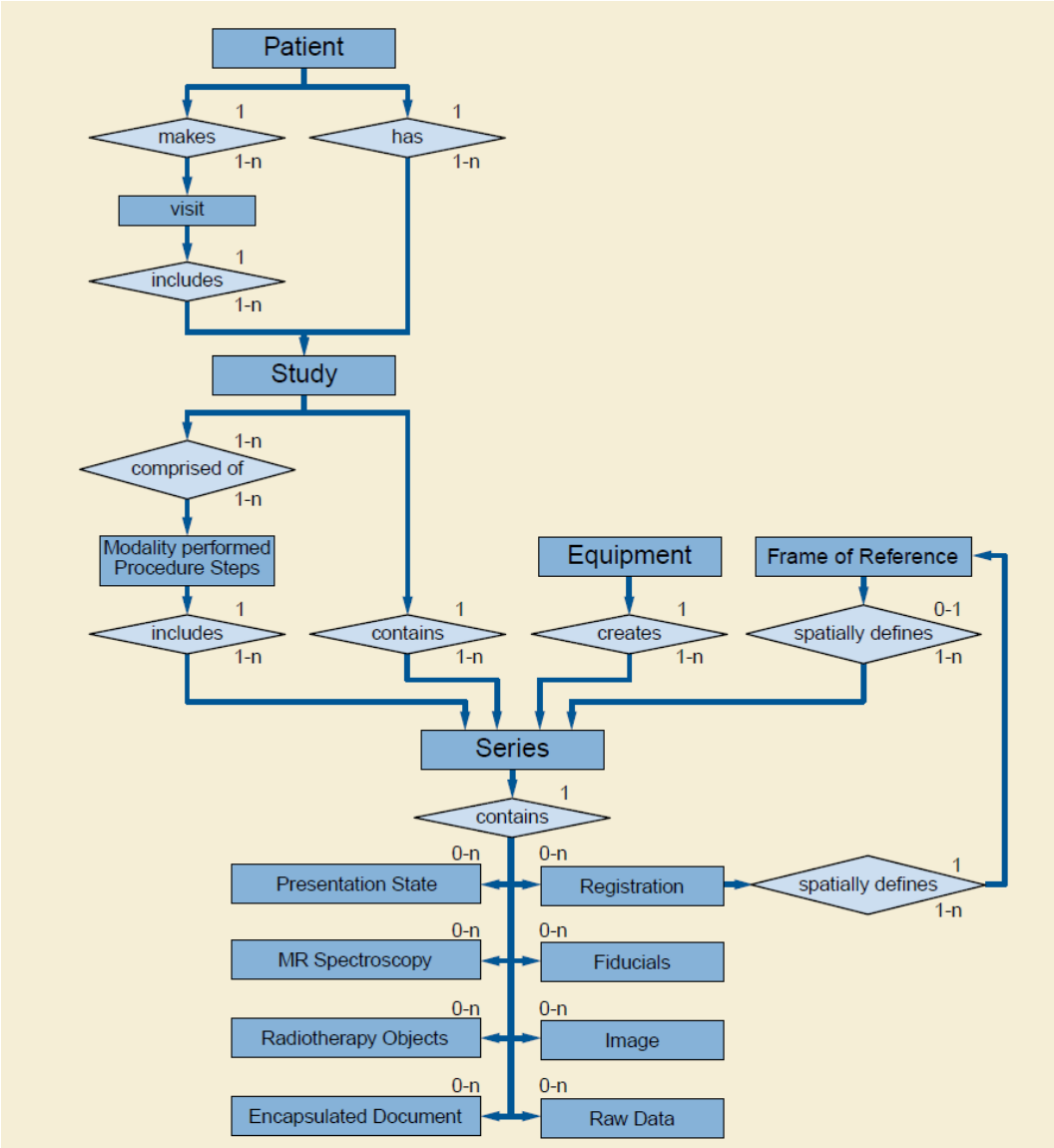
Złożone IOC mogą dodatkowo zawierać atrybuty pośrednio związane z samym obiektem rzeczywistym, który opisują. Na przykład IOC Computed Tomography Image opisujący zdjęcie z tomografu komputerowego, który zdefiniowany jest, jako złożony IOC, zawiera zarówno atrybuty bezpośrednio związane z rzeczywistym obiektem, takie jak data zdjęcia (Image Date) jak i atrybuty 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 DICOM 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,
- dane 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 DICOM (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 DICOM 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 spójności 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:

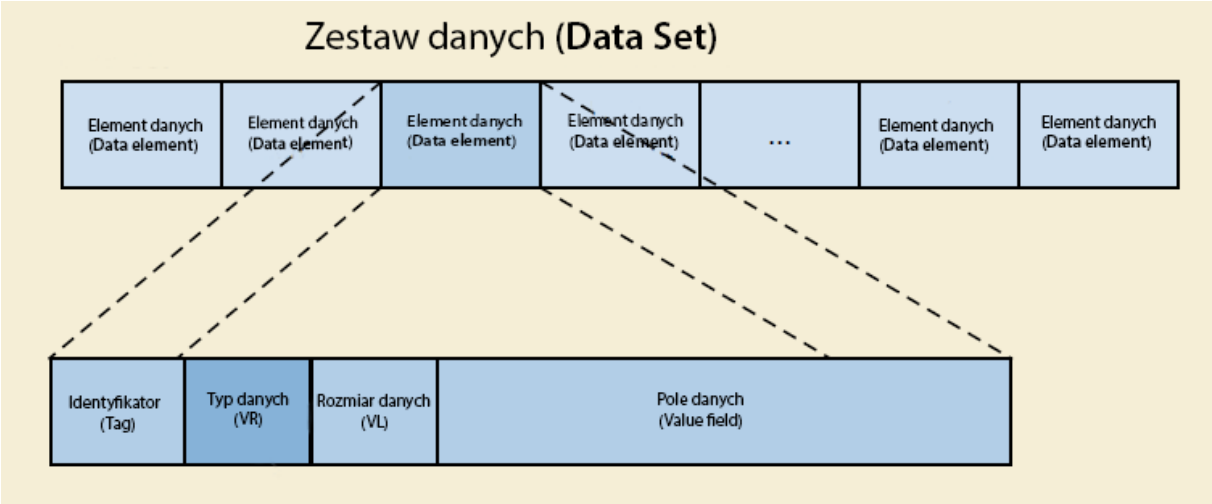
- nagłówek, to jest część zawierającą informacje o pliku DICOM (Dicom-Meta-Information-Header),
- dane obiektu (Dicom-Data-Set).

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 na przykład pojedynczy przekrój z tomografu komputerowego (CT), rezonansu magnetycznego (MRI), czy opis zawartości nośnika, tak jak ma to miejsce w przypadku pliku DICOMDIR.

Podstawową jednostką danych w standardzie DICOM 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. Pola te 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 elementu 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 typy 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.

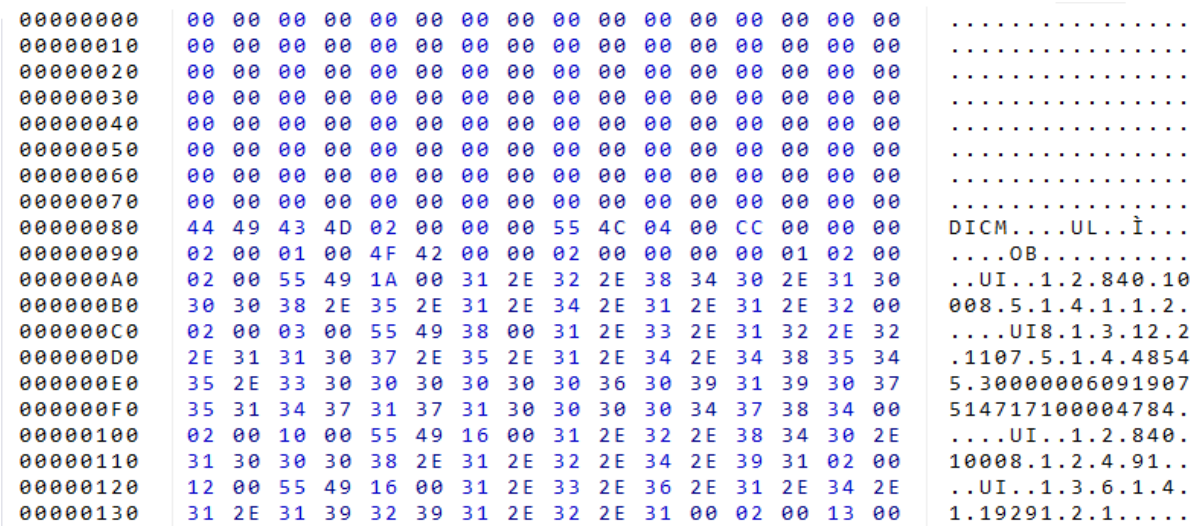
Nielimitowane znaki *Unlimited Characters*	UC	Łańcuch znaków, który może zawierać nieograniczoną długość znaków w tym spacji. Łańcuch nie powinien zawierać znaków sterujących za wyjątkiem znaku ESC.	maks. 2^{32} – 2 B
Unikalny identyfikator *Unique Identyfier*	UI	Łańcuch znakowy zwierający unikalny identyfikator. 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.

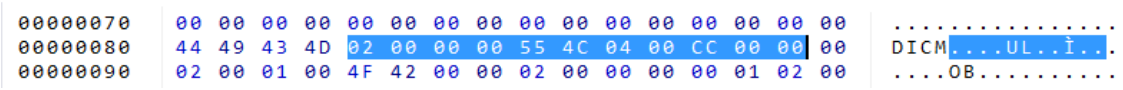
Poniżej na rys 2.8. przedstawiony został plik DICOM z rys 2.2. (wersja z tomografu komputerowego) odczytany w programie HexEdit − tryb szesnastkowy, słowo 8-bitowe.



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

Całość została zgrupowana w trzy kolumny, pierwsza reprezentuje adres bajtu w pliku, 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 ośmio 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 węższą 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 DICOM, 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 pod 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 języków 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 DICOM, 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),
- oprogramowanie przetwarzające zdjęcia pochodzące ze źródeł,
- bazy danych z obsługą zdjęć cyfrowych,
- podsystem umożliwiającego wprowadzanie danych pacjentów,
- podsystem odpowiedzialnego za archiwizację zdjęć cyfrowych,
- podsystem umożliwiającego komunikację to znaczy przesyłanie zdjęć cyfrowych pomiędzy użytkownikami,
- podsystem zapewniającego bezpieczeństwo danych przed nieuprawnionym dostępem.

Idea 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-ciu 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 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 DICOM, 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 podczas 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.

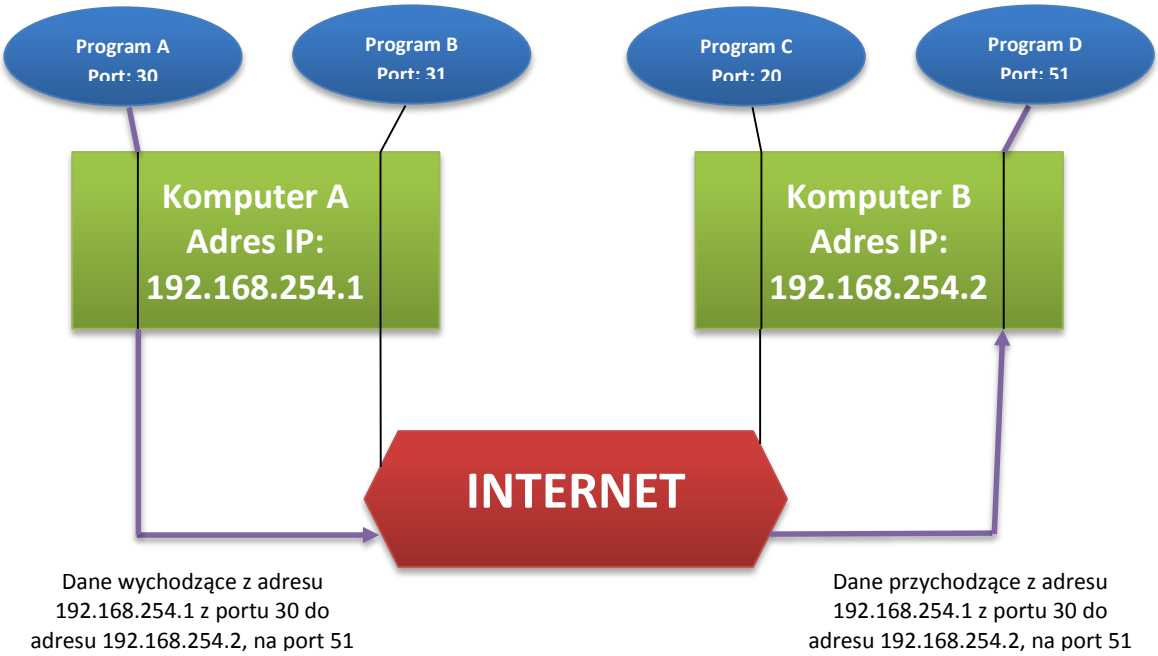
Rys.4.1. Ogólny schemat systemu wizualizacji danych medycznych DICOM.

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.

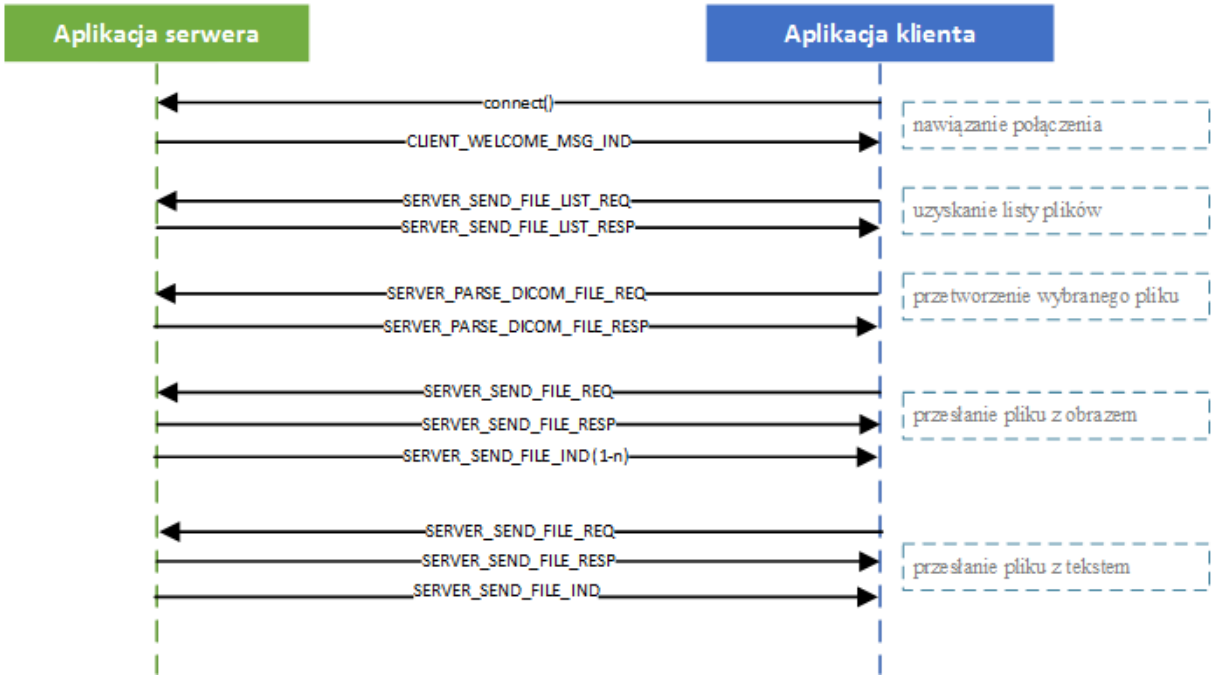
Kolejne pole to ilość bajtów znaczących w polu danych (*bytesInPayload*). Informacja niesiona przez to pole przydatna jest podczas odczytywania pola danych po odebraniu wiadomości. Pole to ma rozmiar 5 bajtów. Maksymalny rozmiar pola danych to 1024 bajty, co można zapisać zgodnie z konwencją ASCII na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

Ostatnim polem jest pole danych (*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).

4.4. Diagram sekwencji dla całego systemu wizualiazcji.

Na rys 4.5 zamieszczono diagram sekwencji dla całego sytemu wizualizacji danych DICOM. Diagram przedstawia wymianę informacji pomiędzy jednostkami systemu, którymi są aplikacja serwera oraz aplikacja kliencka. Celem sytuacji przedstawionej na diagramie jest odczyt wybranego przez użytkownika pliku na aplikacji klienckiej DicomMobile. Pierwszą operacją jest nawiązanie połączenia z serwerem poprzez funkcję *connect*(). Serwer odpowiada wiadomością powitalną CLEINT_WELCOME_MSG_IND.



Rys.4.5. Diagram sekwencji dla systemu wizualizacji.

Następnie uzyskiwana jest lista plików dostępnych na serwerze. Operacja ta realizowana jest za pomocą wymiany wiadomości SERVER_SEND_FILE_LIST_*. Kolejno użytkownik aplikacji klienckiej wybiera plik z listy i do serwera wysyłane jest żądanie przetworzenia wybranego pliku. W odpowiedzi − SERVER_PARSE_DICOM_FILE_RESP zawarta jest lista dwóch plików wynikowych. Pierwszy plik jest to plik binarny zawierający wyodrębniony obraz, drugi plik jest plikiem tekstowym zawierającym dane wyodrębnione z tagów pliku DICOM.

Dalej w dwóch blokach po trzy wiadomości SERVER_SEND_FILE_* realizowane jest przesłanie plików wynikowych. W pierwszej kolejności przesyłany jest plik binarny zawierzający obraz, następnie plik tekstowy. Wymiana pierwszych dwóch wiadomości *_REQ oraz *_RESP określa w ilu częściach będzie przesłany dany plik. Ostatnia wiadomość *_IND zawiera już części wybranego pliku. Zwykle do przesłania pliku binarnego wykorzystywanych jest wiele wiadomości *_IND stąd oznaczenie 1-n na diagramie.

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 potomny, dodatkowo proces macierzysty 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 plików 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);
}
```

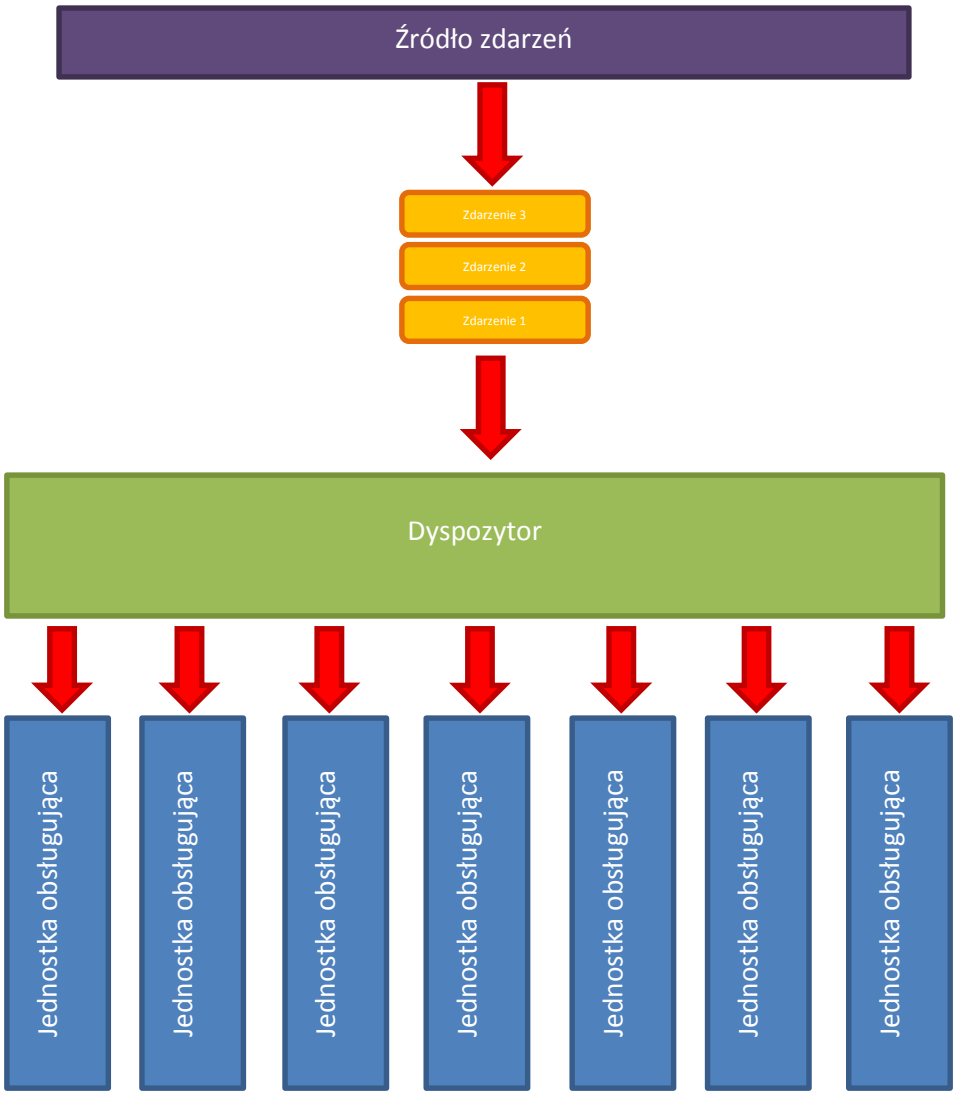
Rys.5.2. Klasa testowa DisaptcherTestSuite wraz z przykładowym testem jednostkowym

```
/*****************************************************************************
 * 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 wewnątrz 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 pole danych (*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. Wiadomość ta posiada między innymi swój identyfikator oraz pole danych. Wiadomość jest odczytywana z gniazda, następnie dekodowana i przekazywana 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 języku 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, zapisu, 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 oraz zawierał informacje na temat instalacji biblioteki w systemie i

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

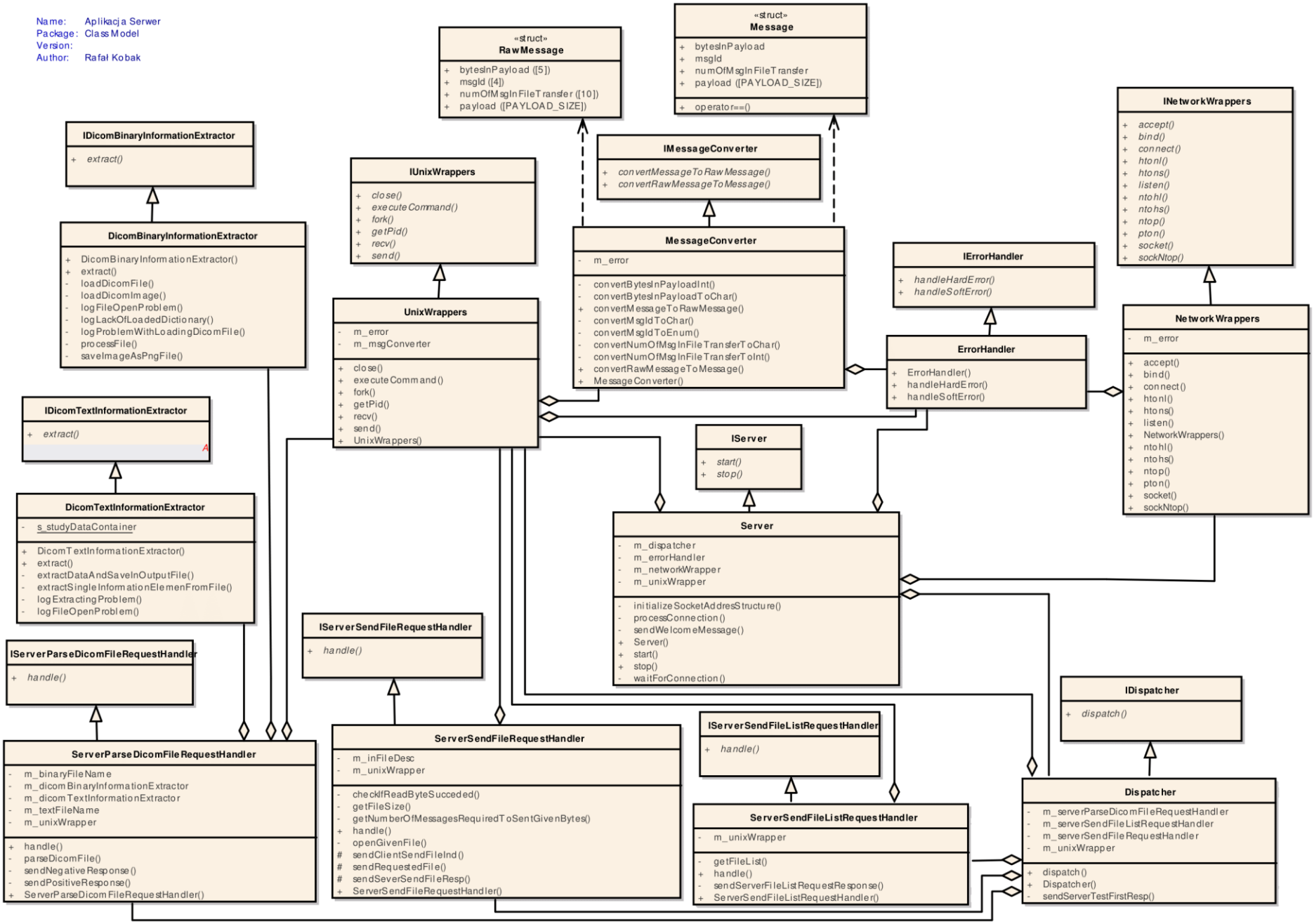
Pakiet dcmdata umożliwił wczytanie samego pliku DICOM do programu. W tym celu niezbędne było dołączenie pliku nagłówkowego dctk.h z tego pakietu. W pakiecie tym znajdują się definicja klasy *DcmFileFormat* oraz implementacja funkcji odczytującej plik – *loadFile()* wraz z definicjami odpowiednich flag 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 struktury sieciowej *SockAddrIn.* 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 typ *GenericSockAddr* (będący aliasem typu *sockaddr* − *sys/socket.h*) 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. Następnie proces aplikacji dzielony jest na dwa procesy. Dla przychodzącego połączenia tworzony jest nowy proces potomny, 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);

    do
    {
        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)
        {
            m_errorHandler->handleHardError("processConnection: recv error");
        }
    }
    while(l_receivedBytes < 0 && errno == EINTR);
}
```

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

Nowy proces potomny 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 polu danych *payload* 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 sieciowe.

```
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*. Odelegowanie to odbywa się za pomocą metody *dispatch()*. Wynik 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.

```
#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.

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.

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 publicznej *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 pole danych to jest uzyskana wcześniej lista plików przekazana do metody w postaci parametru, oraz wielkość pola danych. Uzupełniona wiadomość wysyłana jest do klienta za pomocą metody publicznej *send()* klasy *UnixWrappers*.

5.5.4. Klasa ServerSendFileRequestHandler.

Podobnie jak poprzednio omawiana klasa jest to klasa jednostki obsługującej jednak tym razem dla wiadomości SERVER_SEND_FILE_REQ. Klasa ma interfejs podobny jak poprzednio omawiana. Klasa oprócz klasy *UnixWrappers*, agreguje dodatkowo obiekt klasy *StreamWrapper*, który jest opakowaniem dla klasy obsługującej strumień wejścia i wyjścia. Zawartość pliku nagłówkowego klasy *ServerSendFileRequestHandler* przedstawiona została na rys.5.19.

```
#include "IServerSendFileRequestHandler.hpp"
#include "IUnixWrappers.hpp"
#include "IStreamWrapper.hpp"
#include "StreamWrapper.hpp"
#include "CommonTypes.h"
#include <memory>
#include <string>

#pragma once

class ServerSendFileRequestHandler : public IServerSendFileRequestHandler
{
public:
    ServerSendFileRequestHandler(
        std::shared_ptr<IUnixWrappers> p_unixWrapper,
        std::shared_ptr<IStreamWrapper> p_streamWrapper = std::make_shared<StreamWrapper>());

    void handle(int p_clientSocket, const Message& p_receivedMsg) const override;

protected:
    void sendSeverSendFileResp(int& p_clientSocket,
                               unsigned long long p_fileLength) const;
    void sendRequestedFile(int& p_clientSocket) const;
    void sendClientSendFileInd(Message& p_sendMessage,
                               unsigned long long p bytesInPayload,
                               int& p clientSocket) const;

private:
    void openGivenFile(const std::string& p_filePath) const;
    unsigned long long getFileSize(const std::string& p_filePath) const;
    unsigned int getNumberOfMessagesRequiredToSentGivenBytes(
                                            unsigned long long p_numOfBytes) const;
    void checkIfReadByteSucceded(int p_byteCounter) const;

    std::shared_ptr<IUnixWrappers> m_unixWrapper;
    std::shared_ptr<IStreamWrapper> m_inFileDesc;
};
```

Rys.5.19. Zawartość pliku nagłówkowego klasy *ServerSendFileRequestHandler*.

Zadaniem klasy jest przesłanie do klienta pliku czy to tekstowego czy binarnego określonego w wiadomości SERVER_SEND_FILE_LIST_REQ. Zadanie to jest realizowane poprzez jedyną metodą publiczną to jest metodę *handle*(). Ciało tej metody przedstawione zostało na rys.5.20.

W pierwszej kolejności otwierany jest plik za pomocą prywatnej metody *openGivenFile()*. Implementacja metody przedstawiona jest na rys.5.21 i nie wymaga głębszego omówienia. Kolejno sprawdzany jest jego rozmiar otwartego pliku z wykorzystaniem prywatnej metody *getFileSize()*. Metoda to zwraca rozmiar pliku w bajtach, kolejno wyświetlana jest na

standardowym wyjściu stosowna informacja. Ciało tej metody przedstawione zostało podobnie jak *openGivenFile()* na rys.5.21.

```
void ServerSendFileRequestHandler::handle(int p_clientSocket,
                                          const Message& p_receivedMsg) const
{
    openGivenFile(p_receivedMsg.payload);
    auto l_fileLength = getFileSize(p_receivedMsg.payload);
    std::cout << "File size: " << l_fileLength << " bytes" << std::endl;

    sendSeverSendFileResp(p_clientSocket, l_fileLength);
    sendRequestedFile(p_clientSocket);

    m_inFileDesc->close();
}
```

Rys.5.20. Ciało metody publicznej *handle()*.

Kolejno wysyłana jest odpowiedź do klienta w postaci SERVER_SEND_FILE_RESP, następnie przesyłany jest sam plik – *sendRequestedFile()*. Po zakończonym przetwarzaniu deskryptor pliku otwarty w funkcji *openGivenFile()* nie jest już dłużej potrzebny stąd zostaje zamknięty za pomocą metody *close()*.

```
void ServerSendFileRequestHandler::openGivenFile(const std::string& p_filePath) const
{
    m_inFileDesc->clear();
    m_inFileDesc->open(p_filePath.c_str());

    if (!m_inFileDesc->is_open())
    {
        std::cout << "File " << p_filePath
                  << " is not open. Aplication is going to be terminated"
                  << std::endl;
        exit(-1);
    }
}

unsigned long long ServerSendFileRequestHandler::getFileSize(const std::string& p_filePath)
const
{
    m_inFileDesc->seekg(0, m_inFileDesc->end());
    unsigned long long l_fileLength = m_inFileDesc->tellg();
    m_inFileDesc->seekg(0, m_inFileDesc->beg());

    std::cout << "Size of file: " << p_filePath
              << " is equal to: " << l_fileLength
              << " bytes." << std::endl;

    return l_fileLength;
}
```

Rys.5.21. Ciała metod *openGivenFile()* oraz *getFileSize()*.

W metodzie *sendSeverSendFileResp()*, której ciało przedstawione jest na rys.5.22. realizowane jest odesłanie odpowiedzi do klienta. Przygotowywana jest wiadomość o odpowiednim identyfikatorze *msgId*, z pomocą prywatnej metody pomocniczej *getNumberOfMessagesRequiredToSentGivenBytes()*, ustawiana jest wartość pola *numOfMsgInFileTransfer*, informującego o tym w ilu wiadomościach o *msgId* CLIENT_SEND_FILE_IND przesłany zostanie zamówiony plik. Informacja ta jest niezbędna dla aplikacji klienta.

Po przygotowaniu wiadomość wysyłana jest do klienta za pomocą metody *send()* obiektu klasy *UnixWrapper*.

```
void ServerSendFileRequestHandler::sendSeverSendFileResp(int& p_clientSocket,
                                                         unsigned long long p_fileLength)
const
{
    Message l_sendline = {};
    l_sendline.msgId = SERVER_SEND_FILE_RESP;

    l_sendline.numOfMsgInFileTransfer
         = getNumberOfMessagesRequiredToSentGivenBytes(p_fileLength);
    m_unixWrapper->send(p_clientSocket, &l_sendline);
}

unsigned int ServerSendFileRequestHandler
    ::getNumberOfMessagesRequiredToSentGivenBytes(unsigned long long p_numOfBytes) const
{
    if(p_numOfBytes % PAYLOAD_SIZE)
    {
        return p_numOfBytes / PAYLOAD_SIZE + 1;
    }
    else
    {
        return p_numOfBytes / PAYLOAD_SIZE;
    }
}
```

Rys.5.22. Ciała metod *sendSeverSendFileResp()* oraz metody pomocniczej.

Ciało metody realizującej samo przesłanie zamówionego pliku *sendRequestedFile()* przedstawiono na rys.5.23. Początkowo struktura jest zerowana, następnie w pętli while uzupełniana jest bajt po bajcie wartość pola danych *payload*. Każdorazowo sprawdzona jest poprawność odczytanego bajtu w metodzie *checkIfReadByteSucceded()*. Kiedy liczba odczytanych bajtów jest równa rozmiarowi tablicy pola danych, działanie oddelegowywane jest do metody *sendClientSendFileInd(),* która odpowiada jedynie za uzupełnienie pozostałych pól takich jak np. *msgId* i wysłanie wiadomości z wykorzystaniem metody *send()* obiektu klasy *UnixWrappers*. Każdorazowo po wysłaniu wiadomości logowana jest informacja z jej numerem.

Pętla kończy się, gdy odczytany zostaje ostatni bajt pliku. Następnie wysyłana jest ostatnia wiadomość CLIENT_SEND_FILE_IND z zawartą informacją ile bajtów w polu danych *payload* jest znacząca to jest stanowi część przesyłanego pliku (w wcześniejszych wiadomościach cała tablica *payload* zawierała dane będące częścią pliku) i logowana jest informacja o zakończonym transferze pliku. Dodatkowo logowany jest numer identyfikacyjny procesu PID ze względu na łatwą identyfikację przy obsłudze wielu klientów.

```
void ServerSendFileRequestHandler::sendRequestedFile(int& p_clientSocket) const
{
    unsigned int l_msgCounter = 1;
    unsigned long long l_byteCounter = 0;
    Message l_sendline = {};
    memset(&l_sendline, 0, sizeof(l_sendline));

    char l_singleByte;
    while(m_inFileDesc->get(l_singleByte))
    {
        checkIfReadByteSucceded(l_byteCounter);
        l_sendline.payload[l_byteCounter] = l_singleByte;
        l_byteCounter++;

        if(l_byteCounter == PAYLOAD_SIZE)
        {
            std::cout << "Sending message number: " << l_msgCounter << std::endl;
            sendClientSendFileInd(l_sendline, l_byteCounter, p_clientSocket);
            l_msgCounter++;

            l_byteCounter = 0;
            memset(&l_sendline, 0, sizeof(l_sendline));
        }
    }

    if(l_byteCounter)
    {
        std::cout << "Sending message number: " << l_msgCounter << std::endl;
        sendClientSendFileInd(l_sendline, l_byteCounter, p_clientSocket);
        l_msgCounter++;
    }

    std::cout << m_unixWrapper->getPid() << ": Sending of file is done!" << std::endl;
}
```

Rys.5.22. Ciało metody *sendRequestedFile ()*.

5.5.5. Klasa ServerParseDicomFileRequestHandler.

Klasa ta jest klasą jednostki obsługującej *handler* dla wiadomości o identyfikatorze *msgId* SERVER_PARSE_DICOM_FILE_REQ. Z diagramu przedstawionego na rys5.6 można odczytać, że posiada podobny interfejs jak pozostałe klasy jednostek obsługujących oraz agreguje w sobie obiekty klas *UnixWrappers*, *DicomTextInformationExtractor* oraz *DicomBinaryInformationExtractor*. Dodatkowo klasa ta zawiera dwa obiekty klasy *String*, powinny one być dodatkowo zaznaczone na diagramie z rys.5.6 linią ciągłą z wypełnionym rombem jednak dla zachowania czytelności zostały celowo pominięte. Zawartość pliku nagłówkowego omawianej klasy przedstawiona została na rys.5.23.

Co widać na rys.5.23 obiekt wymaga do utworzenia przekazania w konstruktorze trzech obiektów. Można przyjąć, że klasa ta pełni funkcję zarządcy, wyciąganie konkretnych danych czy to tekstowych czy binarnych (obrazów) oddelegowywane jest do osobnych obiektów.

Ciało metody publicznej *handle* przestawione zostało na rys.5.24. W pierwszej kolejności tworzone są nazwy plików wyjściowych poprzez dodanie przyrostków *.txt* oraz *.png* odpowiednio dla plików tekstowych oraz binarnych będących rezultatem przetwarzania pliku DICOM.

```
#include "IServerParseDicomFileRequestHandler.hpp"
#include "UnixWrappers.hpp"
#include "DicomTextInformationExtractor.hpp"
#include "DicomBinaryInformationExtractor.hpp"
#include "CommonTypes.h"
#include <memory>
#include <string>
#include <vector>
#include <utility>

#include "dcmtk/config/osconfig.h"
#include "dcmtk/dcmdata/dctk.h"
#include "dcmtk/dcmimgle/dcmimage.h"

#pragma once

class ServerParseDicomFileRequestHandler : public IServerParseDicomFileRequestHandler
{
public:
    ServerParseDicomFileRequestHandler(
        std::shared_ptr<IUnixWrappers> p_unixWrapper,
        std::shared_ptr<IDicomTextInformationExtractor> p_dicomTextInformationExtractor
         = std::make_shared<DicomTextInformationExtractor>(),
        std::shared_ptr<IDicomBinaryInformationExtractor> p_dicomBinaryInformationExtractor
         = std::make_shared<DicomBinaryInformationExtractor>());

    void handle(int p_clientSocket, const Message& p_receivedMsg) override;
private:
    void parseDicomFile(int p_clientSocket, const char* p_dicomFileName) const;
    void sendPositiveResponse(int p_clientSocket) const;
    void sendNegativeResponse(int p_clientSocket, std::string p_cause) const;

    std::shared_ptr<IUnixWrappers> m_unixWrapper;
    std::string m_textFileName;
    std::string m_binaryFileName;
    std::shared_ptr<IDicomTextInformationExtractor> m_dicomTextInformationExtractor;
    std::shared_ptr<IDicomBinaryInformationExtractor> m_dicomBinaryInformationExtractor;
};
```

Rys.5.23. Zawartość pliku nagłówkowego klasy ServerParseDicomFileRequestHandler.

Następnie sterowanie przechodzi do prywatnej metody *parseDicomFile()*, której ciało przedstawione jest również na rys.5.24. Metoda ta oddelegowuje przetwarzanie wiadomości do obiektów klas *DicomTextInformationExtractor* oraz *DicomBinaryInformationExtractor* za pomocą ich metod publicznych *extract()*.

```
void ServerParseDicomFileRequestHandler::handle(int p_clientSocket,
                                                const Message& p receivedMsg)
{
    std::cout << "Handling message ServerParseDicomFileRequest" << std::endl;
    m_textFileName = std::string(p_receivedMsg.payload) + ".txt";
    m_binaryFileName = std::string(p_receivedMsg.payload) + ".png";

    parseDicomFile(p_clientSocket, p_receivedMsg.payload);
}

void ServerParseDicomFileRequestHandler::parseDicomFile(int p_clientSocket,
                                                        const char* p_dicomFileName) const
{
    bool l_status = m_dicomTextInformationExtractor->extract(p_dicomFileName,
                                                             m textFileName) and
                    m_dicomBinaryInformationExtractor->extract(p_dicomFileName,
                                                               m_binaryFileName);
    if(l_status)
    {
        sendPositiveResponse(p_clientSocket);
    }
    else
    {
        sendNegativeResponse(p_clientSocket, "Error during data extraction");
    }
}
```

Rys.5.24. Ciała metod *handle()* oraz *parseDicomFile()*.

Następnie w zależności do wyników przetwarzania do klienta odsyłana jest pozytywna bądź negatywna wiadomość SERVER_PARSE_DICOM_FILE_RESP. W przypadku pozytywnej wiadomość zawiera w polu danych nazwy dwóch wynikowych plików rozdzielone spacją, natomiast w przypadku negatywnej w polu danych zawarta jest stosowna informacja, której nie można pomylić z nazwą pliku.

5.5.6. Klasa DicomTextInformationExtractor.

Zadaniem omawianej klasy jest wyodrębnienie z pliku DICOM wybranych wiadomości tekstowych oraz zapisanie ich do pliku tekstowego. Klasa *DicomTextInformationExtractor* jak zostało przedstawione na rys.5.5 dziedziczy po klasie czysto abstrakcyjnej będącą interfejsem. Jedyną metodą publiczną jest metoda *extract()*. Klasa ta zawiera jedynie obiekt *s_studyDataContainer* będący wektorem par, który definiuje zbiór wszystkich danych tekstowych, które chcemy wyodrębnić z pliku DICOM. Pojedynczą parę stanowią obiekt klasy *String*, oraz odpowiedni tag typu *DcmTagKey*, który zdefiniowany jest w bibliotece DCMTK.

```
#include "IDicomTextInformationExtractor.hpp"
#include "CommonTypes.h"
#include <memory>
#include <string>
#include <vector>
#include <utility>

#include "dcmtk/config/osconfig.h"
#include "dcmtk/dcmdata/dctk.h"
#include "dcmtk/dcmimgle/dcmimage.h"

#pragma once

class DicomTextInformationExtractor : public IDicomTextInformationExtractor
{
public:
    DicomTextInformationExtractor();

    bool extract(const char* p_dicomFileName,
                 std::string p_textFileName) const override;
private:
    bool extractDataAndSaveInOutputFile(DcmFileFormat& p_fileFormat,
                                        std::string p_textFileName) const;
    bool extractSingleInformationElemenFromFile(DcmFileFormat& p_fileFormat,
                                                std::ofstream& p_textFile,
                                                std::pair<std::string,
                                                          DcmTagKey> p dataElementAndName)
const;

    void logExtractingProblem(std::string p_elementName) const;
    void logFileOpenProblem(std::string p_textFileName) const;

    static std::vector<std::pair<std::string, DcmTagKey>> s_studyDataContainer;
};
```

Rys.5.25. Zawartość pliku nagłówkowego klasy DicomTextInformationExtractor.

Do metody *extract()*, której ciało przedstawione jest na rys.5.26 przekazywane są nazwa pliku DICOM, z którego dane mają być wyodrębnione oraz nazwa pliku wynikowego gdzie dane

mają być zapisane. Metoda ta zwraca wartość typu bool informującą o rezultacie wyodrębniania.

```
bool DicomTextInformationExtractor::extract(const char* p_dicomFileName,
                                            std::string p_textFileName) const
{
    DcmFileFormat l_fileFormat;
    OFCondition l_status = l_fileFormat.loadFile(p_dicomFileName);
    if (not l_status.good())
    {
        std::cerr << "Error: cannot read DICOM file ("
                  << l_status.text() << ")" << std::endl;
        return false;
    }

    if (not extractDataAndSaveInOutputFile(l_fileFormat, p_textFileName))
    {
        return false;
    }
    return true;
}
```

Rys.5.26. Ciało metody publicznej *extract()* klasy *DicomTextInformationExtractor*.

W pierwszej kolejności otwierany jest plik DICOM z wykorzystaniem funkcji bibliotecznych z biblioteki DCMTK. Metoda *loadFile()* obiektu klasy *DcmFileFormat* wczytuje zawartość pliku i zwraca status. Jeżeli wczytanie pliku nie powiodło się wyświetlany jest stosowny komunikat i zwracana jest wartość false z metody *extract()*. W przypadku sukcesu sterowanie przechodzi do metody *extractDataAndSaveInOutputFile()*, której ciało przedstawione zostało na rys.5.27.

```
bool DicomTextInformationExtractor::extractDataAndSaveInOutputFile(
    DcmFileFormat& p_fileFormat,
    std::string p_textFileName) const
{
    std::ofstream l_textFile(p_textFileName);
    if (l_textFile.is_open())
    {
        for(auto l_element: s_studyDataContainer)
        {
            if(not extractSingleInformationElemenFromFile(p_fileFormat,
                                                          l_textFile,
                                                          l_element))
            {
                logExtractingProblem(l_element.first);
                l_textFile.close();

                return false;
            }
        }

        l_textFile.close();
        return true;
    }
    else
    {
        logFileOpenProblem(p_textFileName);
        return false;
    }
}
```

Rys.5.27. Ciało metody prywatnej *extractDataAndSaveInOutputFile()*.

Zadaniem tej metody jest odczyt danych z otwartego już pliku DICOM i zapis do pliku tekstowego o przekazanej nazwie. Pierwszą operacją w metodzie jest otwarcie pliku tekstowego do zapisu z odpowiednią obsługa błędów. Następnie, jeśli wszystko przebiegło

poprawnie dane są wyodrębniane i zapisywane do pliku wynikowego tak długo aż wszystkie elementy, które chcemy z pliku DICOM pozyskać zostaną odczytane i zapisane w wynikowym pliku tekstowym. Wspomniana wyżej operacja odbywa się za pomocą iteracji po wektorze par *s_studyDataContainer*. Pojedynczy element jest obsługiwany za pomocą metody prywatnej *extractSingleInformationElemenFromFile()*, której ciało przedstawiono na rys.5.28.

```
bool DicomTextInformationExtractor::extractSingleInformationElemenFromFile(
    DcmFileFormat& p_fileFormat,
    std::ofstream& p_textFile,
    std::pair<std::string, DcmTagKey> p_dataElementAndName) const
{
    OFString l_patientData;
    if (p_fileFormat.getDataset()->findAndGetOFString(p_dataElementAndName.second,
                                                      l_patientData).good())
    {
        p_textFile << p_dataElementAndName.first
                   << ": " << l_patientData << std::endl;
        return true;
    }
    else
    {
        std::cerr << "Error: cannot access " << p_dataElementAndName.first
                  << "!" << std::endl;
        return false;
    }
}
```

Rys.5.28. Ciało metody prywatnej *extractSingleInformationElemenFromFile().*

Metoda ta, jako argumenty wywołania przyjmuje otwarte pliki źródłowy DICOM, wynikowy plik tekstowy oraz konkretny tag, który ma wyodrębnić. Za pomocą metody klasy *DcmFileFormat* o nazwie *getDataSet()* pobierany jest zestaw danych i w tym zestawie za pomocą kolejnej metody wspomnianej klasy *findAndGetOFString()* wyszukiwany jest konkretny tag a jego wartość zapisywana jest w obiekcie klasy *OFString*. Klasa *OFString* jest klasą, która również jest zdefiniowana i dostarczona przez bibliotekę DCMTK.

Wspomniana metoda *findAndGetOFString()* zwraca wartość typu bool wskazującą na rezultat operacji, w przypadku niepowodzenia wypisywany jest w programie stosowny log i zwracana wartość false. W przypadku powodzenia pozyskane dane zapisywane są do wynikowego pliku tekstowego za pośrednictwem obiektu klasy *std::ofstream* w ten sposób że każdy element znajduje się w osobnym wierszu gdzie na początku jest wartość z pierwszego elementu pary, to jest obiektu klasy *std::string*, następnie po znaku dwukropka zapisywana jest wartość pozyskana za pomocą funkcji *findAndGetOFString()*.

Przykładowy plik wynikowy będący rezultatem wyodrębniania danych za pomocą obiektu klasy *DicomTextInformationExtractor* przedstawiono na rys.5.29. W razie potrzeby odczytu dodatkowych danych z pliku jedyną wymaganą modyfikacją w klasie jest rozszerzenie wektora o kolejne elementy, reszta zostaje bez zmian.

```
Patient's Name: Perfusion^MCA Stroke
Patient's Sex: M
Patient's Birth Data: 19500704
Patient's Age: 052Y
Patient's Weight: 75
Study Date: 20061219
Study Time: 111154.812
Study Modality: CT
Manufacturer: Acme Medical Devices
DeviceSerialNumber: 123456
Software Version: 1.00
```

Rys.5.28. Tesktowy plik wynikowy będący rezultatem wyodrębniania.

5.5.7. Klasa DicomBinaryInformationExtractor.

Omawiana w tym rozdziale klasa ma podobne przeznaczenie jak poprzednia. Różnicą jest rodzaj danych, które klasa *DicomBinaryInformationExtractor* ma wyodrębnić i w tym przypadku są to dane związane z obrazem, dane binarne. Klasa ta podobnie jak poprzednia posiada klasę bazową deklarująca jedną metodą *extract()*, sama natomiast nie agreguje żadnych obiektów. Plik nagłówkowy klasy *DicomBinaryInformationExtractor* przedstawiony został na rys.5.29.

```
#include "IDicomBinaryInformationExtractor.hpp"
#include "CommonTypes.h"
#include <string>

#include "dcmtk/config/osconfig.h"
#include "dcmtk/dcmdata/dctk.h"
#include "dcmtk/dcmimgle/dcmimage.h"

#pragma once

class DicomBinaryInformationExtractor : public IDicomBinaryInformationExtractor
{
public:
    DicomBinaryInformationExtractor();

    bool extract(const char* p_dicomFileName,
                 std::string p_textFileName) const override;
private:
    bool processFile(FILE*& p_binaryFile, std::string p_fileName) const;
    bool loadDicomFile(DcmFileFormat*& p_dicomFile, std::string p_fileName) const;
    bool loadDicomImage(DicomImage*& p_image, DcmFileFormat*& p_dfile) const;
    void saveImageAsPngFile(FILE*& p_binaryFile, DicomImage*& p_image) const;

    void logFileOpenProblem(std::string p_binaryFileName) const;
    void logLackOfLoadedDictionary() const;
    void logProblemWithLoadingDicomFile(DicomImage* p_image) const;
};
```

Rys.5.29. Zawartość pliku nagłówkowego klasy *DicomBinaryInformationExtractor*.

Jedyną metodą publiczną jest metoda *extract()*, przyjmuje ona jako argumenty wywołania plik źródłowy DICOM oraz nazwę binarnego pliku wynikowego. Metoda zwraca wartość typu bool. Ciało metody przedstawione zostało na rys.5.30. W pierwszej kolejności otwierany jest do zapisu w trybie binarnym plik o otrzymanej nazwie. Będzie to plik wynikowy, do którego zapisywane zostaną dane związane z samym obrazem w pliku DICOM. Podobnie jak w poprzedniej klasie sprawdzany jest rezultat operacji.

```
bool DicomBinaryInformationExtractor::extract(const char* p_dicomFileName,
                                              std::string p_binaryFileName) const
{
    FILE* l_binaryOutputFile = fopen(p_binaryFileName.c_str(), "wb");
    if (l_binaryOutputFile != NULL)
    {
        bool l_status = processFile(l_binaryOutputFile, p_dicomFileName);
        fclose(l_binaryOutputFile);
        return l_status;
    }
    else
    {
        logFileOpenProblem(p_binaryFileName);
        return false;
    }
}
```

Rys.5.30. Ciało metody publicznej *extract()* klasy *DicomBinaryInformationExtractor*.

Jeżeli operacja otwarcia pliku powiedzie się, sterowanie jest przekazywane do prywatnej metody *processFile*() – rys.5.31. W metodzie tej za pomocą metody obiektu globalnego *dcmDatadict* o nazwie *isDictionaryLoaded()* sprawdzane jest czy wewnętrzny słownik biblioteki DCMTK został wczytany. W przypadku odpowiedzi negatywnej wydarzenie to jest logowane i zwracana jest wartość false.

```
bool DicomBinaryInformationExtractor::processFile(FILE*& p_binaryFile,
                                                  std::string p_fileName) const
{
    if (not dcmDataDict.isDictionaryLoaded())
    {
        logLackOfLoadedDictionary();
        return false;
    }

    DcmFileFormat* l_dicomFile = new DcmFileFormat();
    if (not loadDicomFile(l_dicomFile, p_fileName))
    {
        return false;
    }

    DicomImage *l_image = NULL;
    if(not loadDicomImage(l_image, l_dicomFile))
    {
        return false;
    }

    saveImageAsPngFile(p_binaryFile, l_image);
    delete l_image;
    return true;
}
```

Rys.5.31. Ciało metody prywatnej *processFile ()* klasy *DicomBinaryInformationExtractor*.

Następnie podobnie jak w poprzednio omawianej klasie *DicomTextInformationExtractor* za pomocą obiektu klasy *DicomFileFormat* i metody prywatnej klasy *DicomBinaryInformationExtractor* o nazwie *loadDicomFile()* wczytywany do pamięci programu jest źródłowy plik DICOM. Wczytanie pliku w metodzie *loadDicomFile()* odbywa się z wykorzystaniem metody bibliotecznej *loadFile*() klasy *DicomFileFormat* – rys.5.32. Rezultat operacji ponownie jest sprawdzany.

Kolejnym krokiem jest wyodrębnienie z załadowanego pliku bajtów obrazu. Operacja jest realizowana za pomocą zdefiniowanego w bibliotece DCMTK obiektu klasy *DicomImage*

oraz metody prywatnej *loadDicomImage*() przyjmującej wskaźnik na załadowny obraz oraz wskaźnik na obiekt klasy *DicomImage*.

```
bool DicomBinaryInformationExtractor::loadDicomFile(DcmFileFormat*& p_dicomFile,
                                                    std::string p_fileName) const
{
    OFCondition l_result = p_dicomFile->loadFile(p_fileName.c_str(),
                                                 EXS_Unknown,
                                                 EGL_withoutGL,
                                                 DCM_MaxReadLength,
                                                 ERM_autoDetect);
    if (l_result.bad())
    {
        std::cout << __FILE__ << ":" << __LINE__
                  << l_result.text() << ": reading file: " << p_fileName << std::endl;
        return false;
    }
    return true;
}
```

Rys.5.32. Ciało metody prywatnej *loadDicomFile()* klasy *DicomBinaryInformationExtractor*.

Implementacja metody *loadDicomImage()* przedstawiona jest na rys.5.33. Początkowo ustawiane są dane potrzebne do utworzenia obiektu obrazu DICOM – *DicomImage*, ustawiane są stosowne flagi, pobierane są zestawy danych – metody *getDataset*() oraz *getOriginalXfer*(). Następnie na podstawie przygotowanych danych tworzony jest obiekt *DicomImage()* i przypisywany jest do przekazanego w argumencie wywołania wskaźnika. Dodatkowo w funkcji znajduję się obsługa błędów.

```
bool DicomBinaryInformationExtractor::loadDicomImage(DicomImage*& p_image,
                                                     DcmFileFormat*& p_dfile) const
{
    unsigned long l_optCompatibilityMode
        = CIF_MayDetachPixelData | CIF_TakeOverExternalDataset;
    l_optCompatibilityMode |= CIF_IgnoreModalityTransformation;

    DcmDataset* l_dataSet = p_dfile->getDataset();
    E_TransferSyntax l_xfer = l_dataSet->getOriginalXfer();

    OFCmdUnsignedInt l_optFrameCount = 1;
    OFCmdUnsignedInt l_optFrame = 1;

    p_image = new DicomImage(p_dfile,
                             l_xfer,
                             l_optCompatibilityMode,
                             l_optFrame - 1,
                             l_optFrameCount);
    if (p_image == NULL)
    {
        std::cout << __FILE__ << ":" << __LINE__
                  << "Loaded DICOM image is empty!" << std::endl;
        return false;
    }

    if (p_image->getStatus() != EIS_Normal)
    {
        logProblemWithLoadingDicomFile(p_image);
        return false;
    }
    return true;
}
```

Rys.5.33. Ciało metody prywatnej *loadDicomImage()* klasy *DicomBinaryInformationExtractor*.

W tym momencie bajty odpowiedzialne za obraz są załadowane do pamięci programu. Następnie sterowanie zwracane jest do metody *processFile*() i dane te są zapisywane do

popularnego formatu PNG. Operacja ta odbywa się w ciele funkcji *saveImageAsPngFile*(). Metoda ta przyjmuje, jako argumenty wywalania obiekt z wczytanymi danymi obrazu oraz deskryptor do pliku binarnego. Ciało metody przedstawione zostało na rys.5.3.4.

```
void DicomBinaryInformationExtractor::saveImageAsPngFile(
    FILE*& p_binaryFile, DicomImage*& p_image) const
{
    unsigned int l_frame = 0;

    DiPNGPlugin l_pngPlugin;
    l_pngPlugin.setInterlaceType(E_pngInterlaceAdam7);
    l_pngPlugin.setMetainfoType(E_pngFileMetainfo);
    p_image->writePluginFormat(&l_pngPlugin, p_binaryFile, l_frame);
}
```

Rys.5.34. Ciało metody prywatnej *saveImageAsPngFile ()* klasy *DicomBinaryInformationExtractor*.

W wspomnianej metodzie ustawiana jest wartość klatki z obrazu DICOM, która ma być zapisana, tworzony jest obiekt wtyczki dla plików PNG – *DiPNGPlugin* i ustawiane są dane na temat wyników przetwarzania. Dane te są ustawiane za pomocą metod publicznych obiektu klasy *DiPNGPlugin*. Gdy wszystkie wymagane parametry są już ustawione na obiekcie przechowującego wczytany obraz to jest obiekcie klasy *DicomImage* wywoływana jest metoda *writePluginFormat*(), która tworzy plik obrazu w zadanym formacie. Metoda przyjmuje, jako argumenty wskaźnik na obiekt wtyczki, wskaźnik na obiekt z bajtami obrazu oraz numer klatki, który chcemy zapisać w pliku. Po wykonaniu funkcji w określonej przy otwieraniu pliku w lokalizacji znajdować się będzie plik z wyodrębnionym obrazem w formacie PNG. Przykładowy obraz wyodrębniony z pliku DICOM przedstawiony został na rys.5.35.

Rys.5.35. Przykładowy obraz wyodrębniony z użyciem klasy *DicomBinaryInformationExtractor*.

5.5.8. Klasy opakowujące.

W projekcie występują trzy obiekty opakowujące oraz jedna klasa obsługi błędów. Klasami opakowującymi są kolejno *NetworkWrappers*, *UnixWrappers* oraz *StreamWrapper*. Klasy te zostały stworzone z kilku powodów. Dla przykładu klasa *StreamWrapper* w głównej mierze ze względu na możliwość pisania pełnych testów jednostkowych.

Klasy *NetworkWrappers* oraz *UnixWrappers* oprócz wspomnianego powodu powstały również ze względu na przeniesienie obsługi błędów do obiektu tej klasy w celu poprawienia czytelności kodu oraz ułatwienia korzystania z funkcji sieciowych w przypadku klasy *NetworkWrappers* lub systemowych w przypadku funkcji *UnixWrappers*. Przykładowy plik nagłówkowy klasy *UnixWrappers* rys.5.36 zawiera deklaracje metod, które opakowuje, standardowo dziedziczy po klasie interfejsowej. Dodatkowo zawiera obiekty klas *ErrorHandler*, który zostanie po krótce omówiony w tym rozdziale oraz *MessageConverter*.

```
#include "IUnixWrappers.hpp"
#include "ErrorHandler.hpp"
#include "IMessageConverter.hpp"
#include "MessageConverter.hpp"
#include "CommonTypes.h"
#include <memory>
#include <string>

#pragma once

class UnixWrappers : public IUnixWrappers
{
public:
    UnixWrappers(std::shared_ptr<IErrorHandler> p_errorHandler,
                 std::shared_ptr<IMessageConverter> p_msgConverter
                    = std::make_shared<MessageConverter>());

    void send(int p_socketDescriptor,
              const Message* p_messageToSent,
              size_t p_messageLenghtInBytes = sizeof(RawMessage),
              int p transmissionType = 0) const override;

    ssize_t recv(int p_socketDescriptor,
                 Message* p_receivedMessage,
                 size_t p_messageLenghtInBytes = sizeof(RawMessage),
                 int p transmissionType = MSG WAITALL) const override;

    void close(int p_socketDescriptor) const override;
    pid_t fork(void) const override;
    pid_t getPid() const override;
    std::string executeCommand(const char* p_cmd) const override;

private:
    std::shared_ptr<IErrorHandler> m_error;
    std::shared_ptr<IMessageConverter> m_msgConverter;
};
```

Rys.5.36. Zawartość pliku nagłówkowego klasy *UnixWrappers*.

Dla przykładu zostanie omówiona jedna metoda z klas opakowujących, konkretnie metoda *send*() klasy *UnixWrappers*, której ciało przedstawione zostało na rys.5.37. Metoda ta przyjmuje, jako argumenty wywołania podobnie jak metoda, którą opakowuję deskryptor do gniazda sieciowego, wiadomość, która ma być wysłana, (choć innego typu), ilość bajtów do

wysłania oraz zmienną identyfikującą rodzaj transmisji. W pierwszej kolejności dokonywana jest konwersja typu *Message*, który opisuje zdarzenie w programie na obiekt typu *RawMessage*. Obiekt typu *RawMessage* zawiera jedynie pola będące łańcuchami znaków w stylu C a więc pola będące tablicami bajtów. Reprezentacja taka konieczna jest ze względu na przesyłanie wiadomości pomiędzy aplikacjami napisanymi w różnych językach programowania (aplikacja serwera napisana jest w języku C++, natomiast apliakcja kliencka w języku Java).

```
void UnixWrappers::send(int p_socketDescriptor,
                        const Message* p_messageToSent,
                        size_t p_messageLenghtInBytes,
                        int p_transmissionType) const
{
    RawMessage l_rawMsg = m_msgConverter->convertMessageToRawMessage(*p_messageToSent);

    if (::send(p_socketDescriptor,
               &l_rawMsg,
               p_messageLenghtInBytes,
               p_transmissionType) != static_cast<int>(p_messageLenghtInBytes))
    {
        m_error->handleHardError("send error");
    }
}
```

Rys.5.37. Implementacja metody opakowującej *send()* klasy *UnixWrappers*.

Następnie wiadomość jest wysyłana za pomocą systemowej funkcji *send*() i dalej sprawdzany jest rezultat operacji. Jeżeli operacja się powiodła działanie funkcji jest kończone, w przypadku niepowodzenia raportowany jest błąd za pomocą metody *handleHardError*() obiektu *ErrorHandler*.

Klasa *ErrorHandler* służy do wypisywania komunikatów na standardowe wyjście błędów wzbogaconych o dodatkowe informacje. Implementuje dwie metody, *handleHardError()* oraz *handleSoftError()*. W przypadku pierwszej metody oprócz wypisania błędów działanie procesu jest kończone z statusem -1.

5.5.9. Klasa MessageConverter.

Klasa *MessageConverter* dziedziczy po klasie interfejsowej *IMessageConverter*. Klasa ta agreguje obiekt klasy *ErrorHandler* a sama jest agregowana przez klasę *UnixWrappers*. Zadaniem klasy jest konwersja pomiędzy typami wiadomości wykorzystywanymi w systemie to jest *Message* i *RawMessage*. Plik nagłówkowy klasy przedstawiony został na rys.5.38. Klasa ma dwie metody publiczne *convertMessageToRawMessage*(), która przyjmuje, jako argument wywołania obiekt klasy Message i zwraca obiekt klasy *RawMessage* oraz metodę *convertRawMessageToMessage*(), która działa w odwrotny sposób. Obie metody są wykorzystywane każdorazowo przy wysyłaniu oraz odbieraniu wiadomości z gniazda sieciowego.

```
#include "CommonTypes.h"
#include "IMessageConverter.hpp"
#include "ErrorHandler.hpp"
#include <memory>

#pragma once

class MessageConverter : public IMessageConverter
{
public:
    MessageConverter(std::shared_ptr<IErrorHandler> p_errorHandler
                     = std::make_shared<ErrorHandler>());

    RawMessage convertMessageToRawMessage(const Message p_msg) const override;
    Message convertRawMessageToMessage(const RawMessage p_rawMsg) const override;

private:
    void convertMsgIdToChar(const Message& p_msg,
                            RawMessage& p_rawMsg) const;
    void convertNumOfMsgInFileTransferToChar(const Message& p_msg,
                                             RawMessage& p_rawMsg) const;
    void convertBytesInPayloadToChar(const Message& p_msg,
                                     RawMessage& p_rawMsg) const;

    void convertMsgIdToEnum(const RawMessage& p_rawMsg,
                            Message& p_msg) const;
    void convertNumOfMsgInFileTransferToInt(const RawMessage& p_rawMsg,
                                            Message& p_msg) const;
    void convertBytesInPayloadInt(const RawMessage& p_rawMsg,
                                  Message& p_msg) const;

    std::shared_ptr<IErrorHandler> m_error;
};
```

Rys.5.38. Zawartość pliku nagłówkowego klasy *MessageConverter*.

Same metody publiczne tworzą w swoim ciele pusty obiekt wynikowy, następnie za pomocą specjalnych konwerterów uzupełniane jest pole po polu. Wyjątkiem jest pole danych *Payload*, które ze względu na zgodność typów w *Message* i *RawMessage* wymaga jedynie kopiowania z jednej struktury do drugiej. Ciało metody *convertRawMessageToMessage()* przedstawiono na rys.5.39. Konwersja poszczególnych pól odbywa się z wykorzystaniem *boost::lexical_cast*.

```
Message MessageConverter::convertRawMessageToMessage(const RawMessage p_rawMsg) const
{
    Message l_msg = {};

    convertMsgIdToEnum(p_rawMsg, l_msg);
    convertNumOfMsgInFileTransferToInt(p_rawMsg, l_msg);
    convertBytesInPayloadInt(p_rawMsg, l_msg);
    memcpy(l_msg.payload, p_rawMsg.payload, l_msg.bytesInPayload);

    return l_msg;
}
```

Rys.5.37. Implementacja metody *convertRawMessageToMessage()* klasy *MessageConverter*.

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:

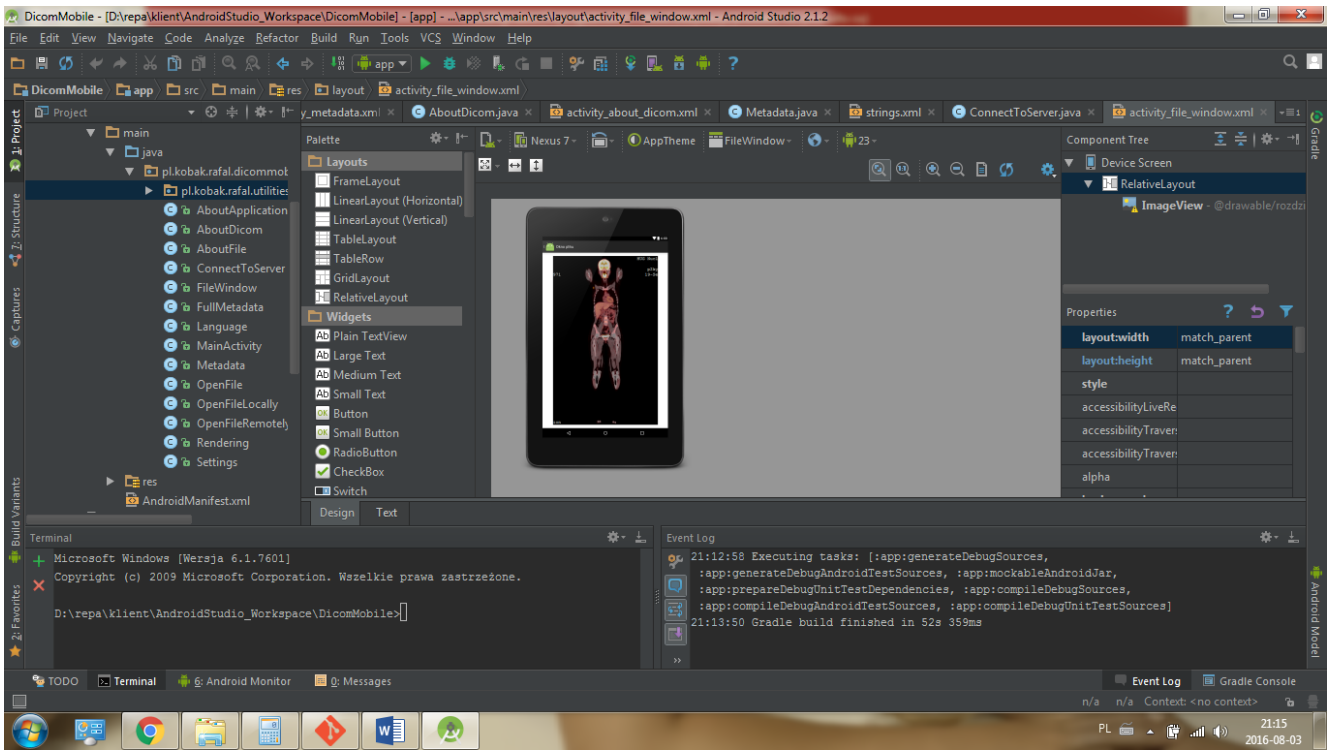
- możliwość połączenia z aplikacją serwera poprzez protokół TCP,
- prezentacja wyników parsowania aplikacji serwera w postaci danych graficznych oraz danych tekstowych,
- działanie na urządzeniach mobilnych typu tablet oraz smartfon.

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

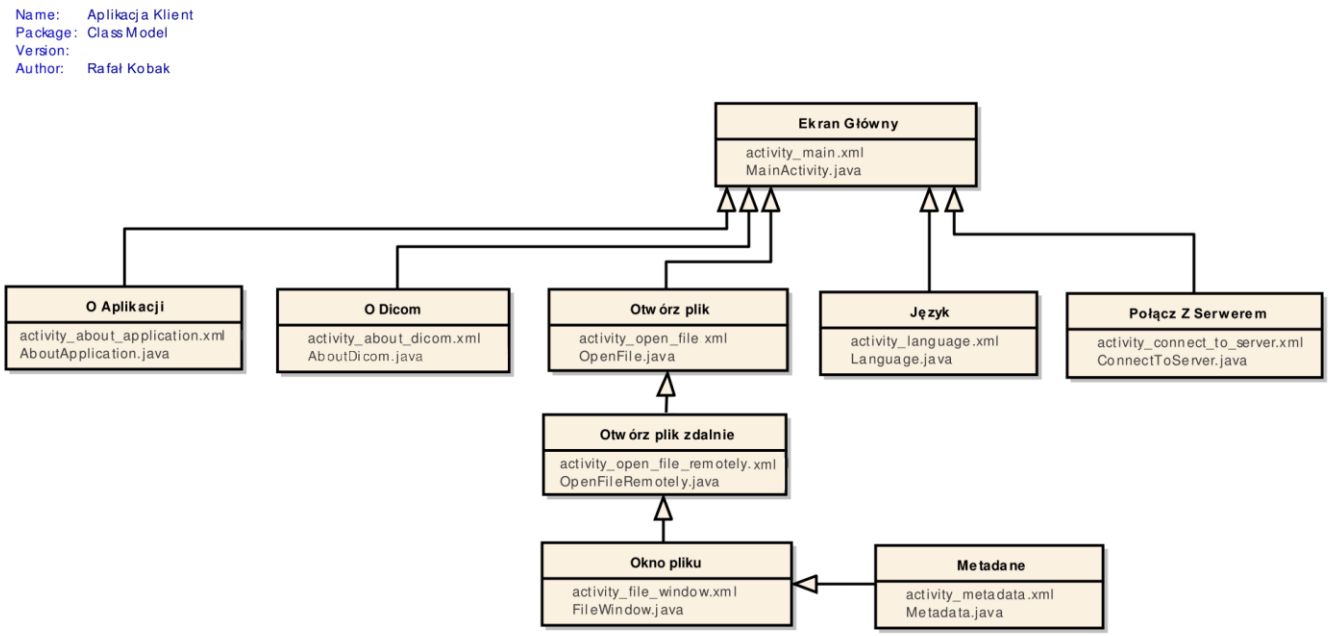
różnych wersji językowych aplikacji. Innymi zasobami aplikacji są pliki *dimens.xml*, w których przechowywane są wymiary używane w aplikacji, na przykład wymiary przycisków, pól tekstowych oraz *styles.xml*, w których przechowywane są style poszczególnych elementów.

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.

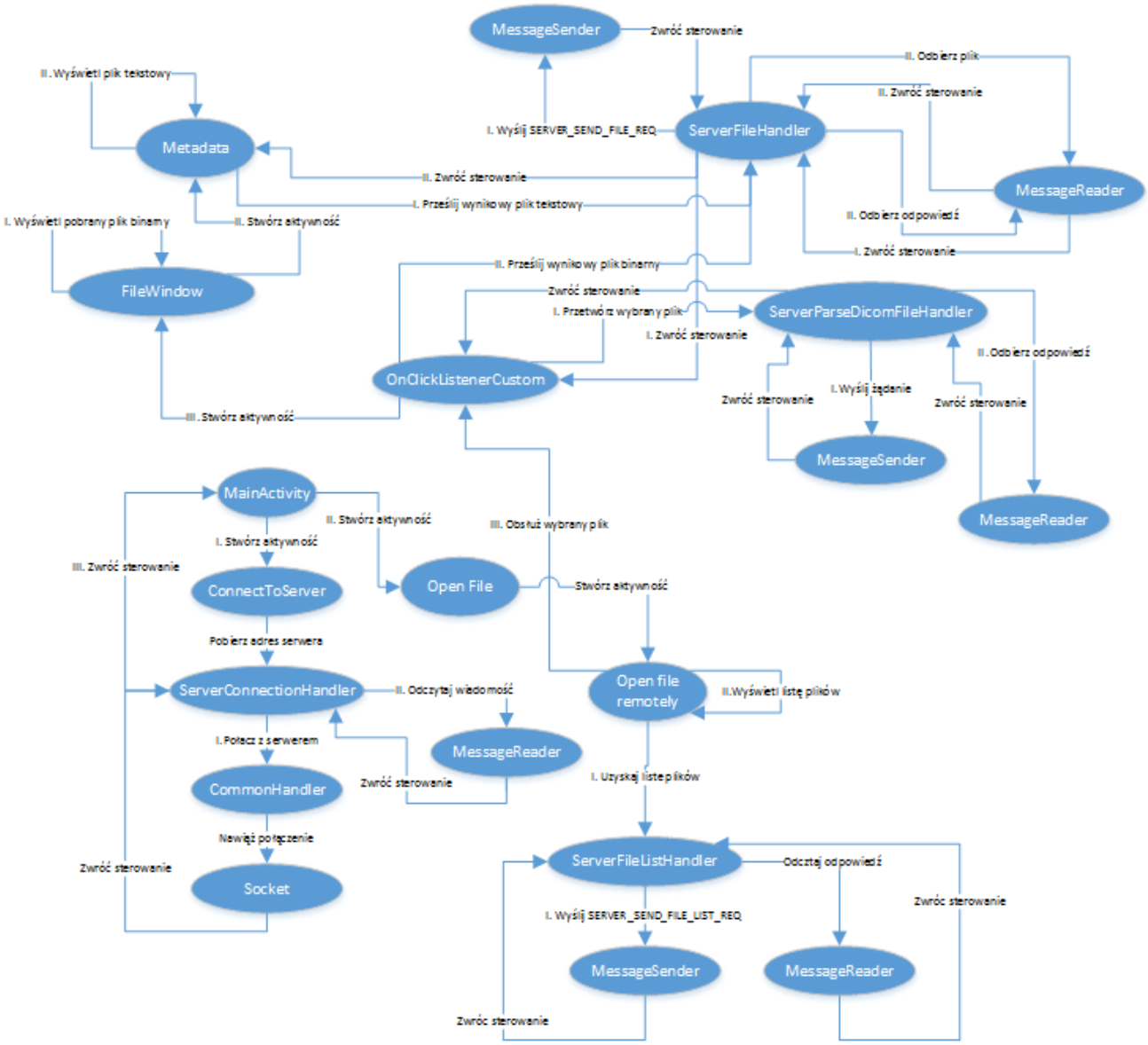
Architektura aplikacji klienckiej przedstawiona została na rys.6.2. Za pomocą strzałek przedstawiono zależności pomiędzy poszczególnymi ekranami to jest ekrany nadrzędne oraz ekrany podrzędne. Ekranem nadrzędnym dla wszsytkich ekranów jest ekran główny aplikacji. Jest to ekran, który pojawia się zaraz po włączeniu aplikacji DicomMobile na tablecie.

Na każdy ekran w aplikacji działającej pod systemem Android stworzonej z wykorzystaniem środowiska Android Studio składają się, co najmniej dwa pliki – plik aktywności w formacie XML oraz plik z kodem źródłowym w języku Java. W pliku XML definiuje się style, oraz elementy graficzne ekranu, natomiast w pliku z rozszerzeniem .java definiuje się zachowania będące odpowiedzią na zdarzenia od użytkownika, na przykład dotknięcie przycisku na ekranie.



Rys.6.2. Diagram przedstawiający ekrany oraz zależności pomiędzy nimi w aplikacji klienkciej.

Na rys.6.3 przedstawiono diagram przepływu dla aplikacji klienckiej. Diagram obrazuje analogiczną sytuację jak diagram 4.5 to jest sytuację, w której użytkownik aplikacjji mobilnej chce połączyć się z serwerem oraz podglądnać zawartość jednego z plikow DICOM udostępnianych przez serwer. Różnica pomiędzy diagramami (poza ich rodzajem) polega na tym, iż diagram z rys.4.5 przedstawia cały system bez wgłębiania się w szczegóły którejkolwiek jednostki systemu natomiast diagram 6.3 przedstawia omawianą sytuację jedynie od strony aplikacji klienta, uwzględniając przy tym przepływ danych w klasach wewnątrz aplikacji.



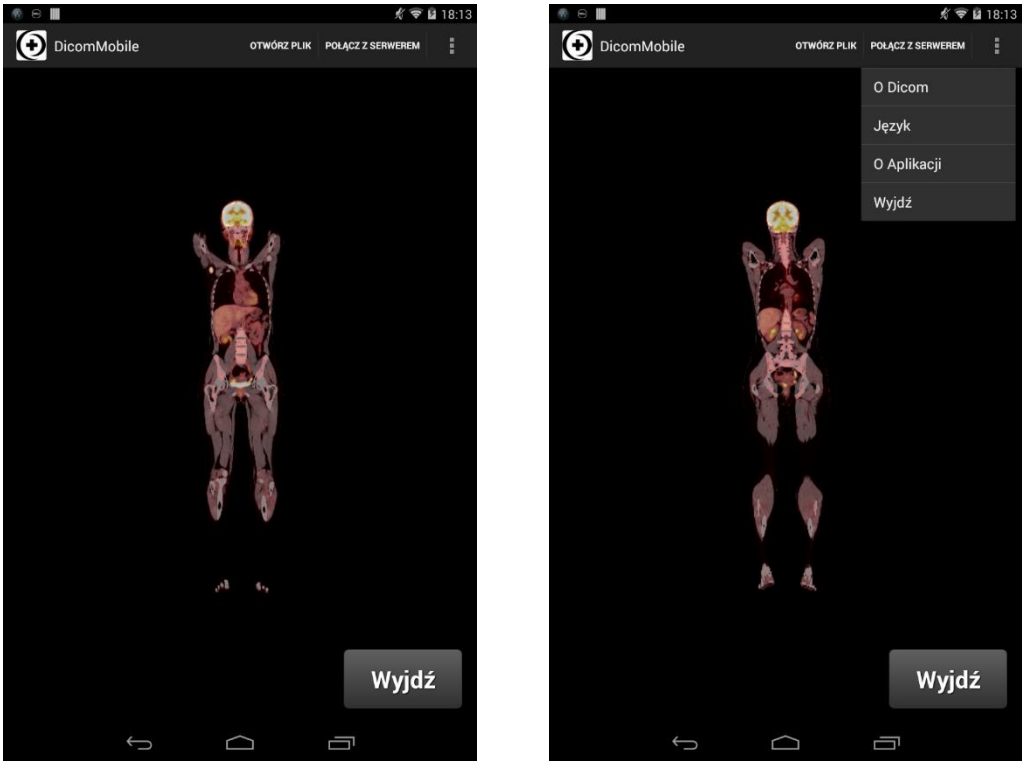
Rys.6.3. Diagram przepływu dla aplikacji klienckiej DicomMobile.

Miejscem startu na diagramie jest blok *MainActivity.* Tutaj pierwszą operacją jest połączenie z serwerem realizowane z wykorzystaniem aktywności *ConnectToServer*. Dalej po połączeniu z serwerem i odczytaniu wiadomości powitalnej tworzona jest aktywność *OpenFile* a dalej *OpenFileRemotely*. Aktywność ta wykonuje trzy operacje. Uzyskuje listę plików DICOM

dostępnych na serwerze za pomocą wymiany wiadomości SERVER_SEND_FILE_LIST_*z aplikacją serwera, wyświetla otrzymaną listę na ekranie a następnie oddelegowuje działanie do klasy *OnClickListenerCustom*. Klasa *OnClickListenerCustom* używa klasy *ServerParseDicomFileHandler* do przetworzenia wybranego przez użytkownika pliku a następnie klasy *ServerFileHandler* do otrzymania od serwera wynikowego pliku graficznego. Dalej wspomniana klasa tworzy nową aktywność *FileWindow*. Nowo utworzona aktywność wczytuje trzymany już lokalnie plik wynikowy z obrazem do programu i wyświetla zawartość na ekranie. Dalej klasa tworzy aktywność *Metadata*, która ponownie z wykorzystniem klasy *ServerFileHandler* uzyskuje wynikowy plik tekstowy. Zawartość tego pliku prezentowana jest następnie na ekranie.

6.3.1. Ekran główny.

Ekran główny uruchamiany jest zaraz po starcie aplikacji. Ekran ten przedstawiony został na w postaci dwóch zrzutów ekranu na rys.6.4. Cały ekran składa się właściwie z trzech elementów, zapętlonej animacji poklatkowej stworzonej z obrazów uzyskanych z pliku DICOM, menu ekranowego umożlwiającego nawigację po aplikacji oraz przycisku wyjdź.



Rys.6.4. Ekran głowny aplikacji z zwiniętym oraz rozwiniętym menu.

W lewym górnym rogu widnieje nazwa oraz ikona aplikacji, dalej w menu możemy przejść bezpośrednio do ekranów potomnych to jest ekranów „Otwórz plik" oraz „Połącz z serwerem". Pozostałe ekrany domyślnie schowane są pod znakiem pionowego trzykropka. Po

kliknięciu na ten znak pokazuje się rozwijana lista umożliwiająca przejście do pozostałych ekranów potomnych.

Zawartość pliku XML opisującego aktywność przedstawiona została na rys.6.5. W ekranie tym zastosowano układ relatywny to znaczy, że położenie kolejnych elementów ustala się za pomocą położenia do elementu już istniejącego. W przypadku pierwszego elementu jego położenie określa się względem ramek samego układu. Dzięki zastosowaniu układu relatywnego położenie elementów zostaje zachowane na różnych ekranach oraz przy dowolnej orientacji ekranu tabletu tak portretowej jak i krajobrazowej.

W tagu *RelativeLayout* określono, że ekran ma być w całości wypełniony układem oraz określono jego tło. W tagu *Button*, który definiuje przycisk „Wyjdź" na ekranie określone są jego rozmiary za pomocą parametrów `android:layout_width` oraz `android:layout_heigth`. Same wartości rozmiarów określone są w specjalnym pliku *dimen.xml* w postaci stałych.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
                xmlns:tools="http://schemas.android.com/tools"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:paddingLeft="@dimen/activity horizontal margin"
                android:paddingRight="@dimen/activity horizontal margin"
                android:paddingTop="@dimen/activity_vertical_margin"
                android:paddingBottom="@dimen/activity_vertical_margin"
                tools:context=".MainActivity"
                android:background="@android:color/background dark">

    <Button
        android:layout_width="@dimen/button_width"
        android:layout_height="@dimen/button_height"
        android:text="@string/action_quit"
        android:id="@+id/button quit"
        android:onClick="onClick quit"
        android:background="@drawable/button"
        style="@style/buttonStyle"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout alignParentEnd="true"
        android:layout alignParentStart="false"
        android:layout_alignParentLeft="false"/>

    <ImageView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:id="@+id/animationViewMain"
        android:background="@drawable/animation_main_window"
        android:layout centerVertical="true"
        android:layout centerHorizontal="true"
        android:onClick="onClickMainAnimation"/>

</RelativeLayout>
```

Rys.6.5. Zawartość pliku *activity_main.xml*.

Kolejne parametry określają kolejno tekst wyświetlany na przycisku, którego wartość została zdefiniowana również za pomocą zmiennej w specjalnym pliku *strings.xml*, następnie

występuje parametr, który określa identyfikator obiektu przycisku w aplikacji klienckiej (umożliwia to dostęp do przycisku z poziomu plików.java).

Ważnym parametrem jest parametr `android:onClick`, określa on, co ma się stać po dotknięciu przycisku na ekranie i tworzy uchwyt do kodu źródłowego ekranu. W opisywanym parametrze w momencie dotknięcia przycisku wywoływana jest metoda publiczna *onClick_quit()* klasy ekranu to jest *MainActivity*. Kolejnymi parametrami są parametry określające kolejno tło przycisku oraz styl przycisku zdefiniowany również w osobnym pliku *styles.xml*. Dalsze parametry określają położenie elementu przycisku na ekranie.

Kolejnym tagiem jest tag *ImageView*, który jest w opisywanym ekranie tagiem animacji. Parametr `android:background` wskazuje położenie pliku obrazu które stanowi tło elementu *ImageView*. Warto zaznaczyć, że wszystkie elementy graficzne dostarczane z zewnątrz do aplikacji tworzonej w systemie android muszą znaleźć się w folderze *drawable* projektu. Pozostałe parametry zostały omówione przy poprzednim tagu.

Na rys.6.6. przedstawiono ciało metody publicznej *onCreate().* Metoda ta jest wywoływana za każdym razem, gdy ekran jest otwierany. W pierwszej linii wywoływana jest metoda *onCreate()* klasy bazowej dla klasy *MainActivity* to jest klasy *Activity* definiującą aktywność. Następnie za pomocą metody *setContentView*() zawartość ekranu ustawiana jest ustawiana na podstawie pliku *activity_main.xml* opisywanego kilka linijek wyżej. Dostęp do pliku XML odbywa się z wykorzystaniem identyfikatora *R.layout.activity_main*.

```
@Override
protected void onCreate(Bundle p savedInstanceState)
{
    super.onCreate(p_savedInstanceState);
    setContentView(R.layout.activity_main);
}

public static Socket s_socket;
public static String s_ipAddress;
public static String s_portNumber;
public static String s_fileList;
public static String s_chosenFileName;
public static String s welcomeMessage;

private boolean m_isMovement = false;
```

Rys.6.6. Metoda *onCreate()* klasy *MainActivity* oraz obiekty statyczne wykorzystywane w aplikacji.

Dalej w pliku zestawione są wszystkie obiekty statyczne, to jest obiekty istniejące przez cały czas działania programu. Ostatnią linijką jest definicja prywatnej zmiennej składowej wykorzystywanej przez opisywaną klasę.

```
public void onClickMainAnimation(View p_view)
{
    ImageView l_img = (ImageView) findViewById(R.id.animationViewMain);
    AnimationDrawable l_animation = (AnimationDrawable) l_img.getBackground();

    if (m_isMovement)
    {
        l_animation.stop();
        m_isMovement = false;
    }
    else
    {
        l_animation.start();
        m_isMovement = true;
    }
}
```

Rys.6.7. Implementacja metody *onClickMainAnimation()*.

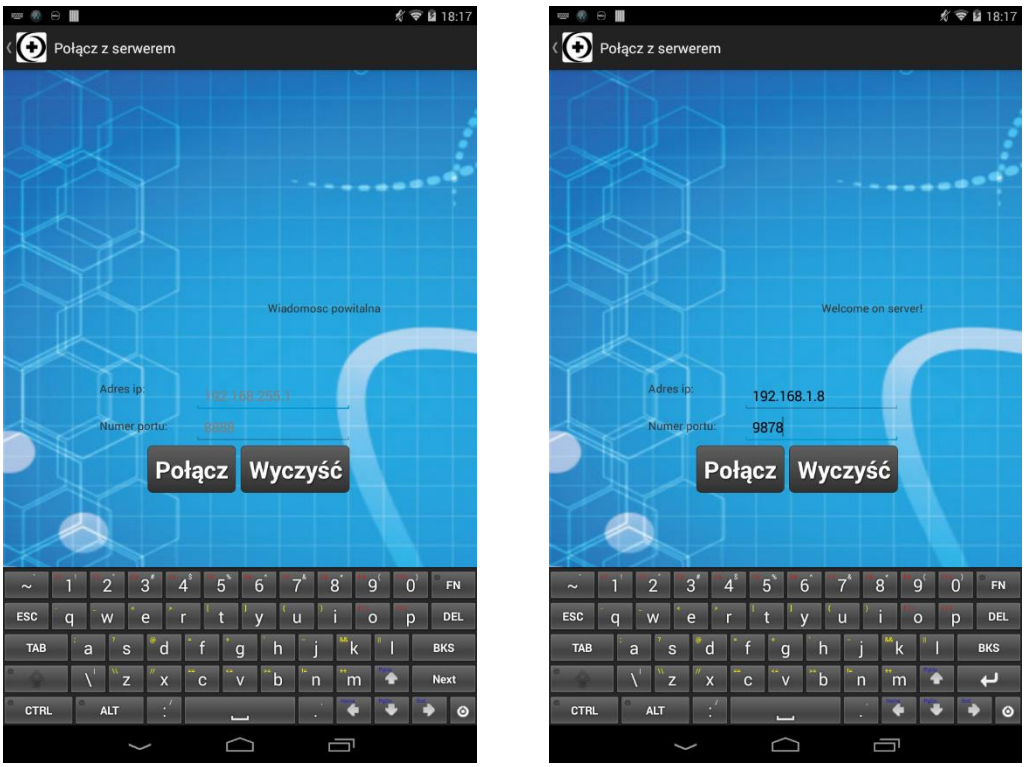
Na rys.6.7 przedstawiono implementację metody *onClickMainAnimation()*. Metoda ta jest wywoływana każdorazowo w momencie dotknięcia ekranu w miejscu gdzie widnieje obiekt *ImageView*. W pierwszej linii metody, pobierany jest za pomocą identyfikatora i funkcji *findViewById()* obiekt *ImageView*, następnie pobierane jest tło uzyskanego obiektu. Tłem tym inicjalizowany jest obiekt klasy *AnimationDrawable*. Następnie w zależności od wartości flagi *m_isMovement* animacja jest startowana bądź zatrzymywana z wykorzystaniem swoich metod publicznych *start()* i *stop()*.

Sama definicja bieżącej animacji, to jest określenie takich parametrów jak obrazy wchodzące w skład animacji poklatkowej lub czas wyświetlania pojedynczego obrazu w animacji określone są w pliku *animation_main_window.xml*. Plik ten znajduje się w folderze *drawable* projektu.

6.3.2. Ekran „Połącz z serwerem”.

Ekran „Połącz z serwerem” zawiera więcej elementów niż poprzednio omawiane ekrany, jego budowę przedstawiono w postaci zrzutów ekranu na rys.6.8. „Połącz z serwerem” jest bezpośrednim ekranem potomnym ekranu „Ekran główny”. Ekran ten zawiera pola tekstowe, przyciski oraz tak zwane pola edycji. „Połącz z serwerem” zgodnie z nazwą służy do nawiązania połączenia z serwerem. W pola edycyjne należy wpisać adres IP serwera oraz numer portu, na którym aplikacja serwera nasłuchuje. Po wpisaniu danych należy nacisnąć przycisk połącz. W przypadku pomyłki w adresie czy też numerze portu można skorzystać z przycisku wyczyść.

Po połączeniu z serwerem w przypadku sukcesu tekst w polu tekstowych ponad polami edycyjnymi zmieni się z „Wiadomość powitalna” na „Welcome on server!”.



Rys.6.8. Ekran „Połącz z serwerem" przed i po połączeniu.

Klasa źródłowa to jest klasa *ConnectToServer* standardowo dla klas aktywności dziedziczy po klasie *Activity*. Obiekty składowe klasy źródłowej, konstruktor oraz metoda publiczna *onCreate()* zostały przedstawione na rys.6.9.

```
final String LABEL = getClass().getSimpleName();
private IpAddressValidator m_addressValidator;

public ConnectToServer()
{
    super();
    m_addressValidator = new IpAddressValidator();
}

@Override
protected void onCreate(Bundle p savedInstanceState)
{
    super.onCreate(p_savedInstanceState);
    setContentView(R.layout.activity_connect_to_server);
}
```

Rys.6.9. Deklaracje obiektów składowych oraz definicje metody *onCreate()* oraz konstruktora.

Obiektami składowymi klasy jest obiekt *LABEL* będący identyfikatorem klasy na potrzeby logowania oraz obiekt klasy *IpAddressValidator*, którego zadaniem jest sprawdzanie poprawności wprowadzonego adresu IP oraz numeru portu. Klasa *IpAddressValidator* omówiona zostanie w dalszej części rozdziału. W konstruktorze inicjalizowane są obiekty składowe oraz wywoływany jest konstruktor klasy bazowej *Activity*. Dalej w metodzie *onCreate*() jedyną operacją jest operacja stworzenia ekranu za pomocą metody *setContentView*() oraz pliku *activity_connect_to_server.xml*.

```
public void onClick_clear(View p_view)
{
    EditText l_ipAddressEditText = (EditText) findViewById(R.id.ipAddressTextEdit);
    l_ipAddressEditText.getText().clear();
    l_ipAddressEditText.setHint(getString(R.string.hint_ipAddress));

    EditText l_portNumberEditText = (EditText) findViewById(R.id.portNumberEditText);
    l_portNumberEditText.getText().clear();
    l_portNumberEditText.setHint(R.string.hint_portNumber);
}
```

Rys.6.10. Definicja metody *onClick_clear().*

Na rys.6.10 przedstawiono implementacje metody *onClick_clear*(). Metoda ta jest uchwytem na przycisk „Wyczyść". Dotknięcie przycisku powoduje, więc wywołanie powyższej metody. Pierwszą operacją w metodzie jest uzyskanie dostępu do pól edycyjnych przeznaczonych do wprowadzenia adresu IP oraz numeru portu. Dostęp ten odbywa się z wykorzystaniem identyfikatora. Następnie zawartość tych pól jest czyszczona oraz wyświetlana jest wyszarzona podpowiedź.

Na rys.6.11 przedstawiono ciało metody będącej uchwytem na przycisk „Połącz" to jest *onClick_connect*(). W pierwszej kolejności w metodzie uzyskiwany jest dostęp do pól edycyjnych służących do wprowadzania adresu IP oraz numeru portu, następnie pobierany jest z nich wprowadzony przez użytkownika tekst i zapisywany jest do lokalnych obiektów klasy *String*. Wprowadzone są weryfikowane pod względem poprawności składniowej za pomocą obiektu klasy *IpAddressValidator*.

```
public void onClick_connect(View p_view)
{
    EditText l_ipAddressEditText = (EditText) findViewById(R.id.ipAddressTextEdit);
    String l_ipAddress = l_ipAddressEditText.getText().toString();

    EditText l_portNumberEditText = (EditText) findViewById(R.id.portNumberEditText);
    String l_portNumber = l_portNumberEditText.getText().toString();

    logReceivedAddress(l_ipAddress, l_portNumber);
    boolean l_isValid = m_addressValidator.validateServerAddress(l_ipAddress, l_portNumber);
    if (!l_isValid)
    {
        Log.d(LABEL, "Invalid ip address or port number!");

        l_ipAddressEditText.getText().clear();
        l_ipAddressEditText.setHint(getString(R.string.incorrectData));
        l_portNumberEditText.getText().clear();
        l_portNumberEditText.setHint(getString(R.string.incorrectData));

        return;
    }

    MainActivity.s_ipAddress = l_ipAddress;
    MainActivity.s_portNumber = l_portNumber;

    Thread l_connectionThread = new Thread(new ServerConnectionHandler());
    l_connectionThread.start();
    while(l_connectionThread.isAlive()) {}

    TextView l_textView = (TextView) this.findViewById(R.id.welcomeMessageTextView);
    l_textView.setText(MainActivity.s_welcomeMessage);
}
```

Rys.6.11. Definicja metody *onClick_connect().*

W przypadku, gdy wprowadzony adres IP lub numer portu jest niepoprawny informacja ta logowana jest w strumieniu błędów, następnie pola edycyjne są czyszczone i wypełniane tekstem informującym użytkownika o niepoprawnych danych wejściowych. Należy wtedy ponownie wprowadzić adres IP oraz numer portu. W przypadku, gdy dane są poprawne, zostają one zapisane w zmiennych statycznych *s_ipAdress* oraz *s_portNumber* klasy *ManActivity*. Przypisania takie są możliwe, ponieważ oba pola są polami publicznymi.

Następnie tworzony jest obiekt wątku dla obsługi połączenia z serwerem. Wątek ten inicjalizowany jest obiektem klasy *ServerConnectionHandler()* i linijkę niżej startowany. Następnie z pomocą pętli while oczekujemy na zakończenie procedury nawiazania polaczenia z serwerem. Po zakończeniu tej procedury zawartość pola tekstowego *welcomeMessageTextView* uzupełniana jest wiadomościa powitalną otrzymaną z serwera za pośrednictwem obiektu statycznego *s_welcomeMessage* klasy *ManActivity* uzupełnionego wcześniej przez obiekt klasy *ServerConnectionHandler.*

Na omówienie zasługije metoda *validateServerAdress* klasy *IpAddressValidator*. Metoda ta zwraca wartość typu boolean informującą użytkownika o tym czy adres oraz numer portu są poprawne czy nie. W przypadku numeru portu sprawdzane jest czy zmienna nie jest pusta oraz czy zawiera tylko cyfry. W przypadku adresu IP weryfikowane jest czy zmienna nie jest pusta, czy nie kończy się na znakiem '.' oraz czy składa się z czterech ciągów cyfr rozdzielonych znakiem ', ' gdzie każdy ciąg powinien być zbudowany z cyfr z zakresu od 0 do 255. W przypadku, gdy choć jeden warunek nie jest spełniony metoda zwraca wartość false.

Na rys.6.12 przedstawiono definicję klasy *ServerConnectionHandler*. Zadaniem tej klasy jest nawiązanie połaczenia z aplikacją serwera. Klasa ta dziedziczy po klasie *CommonHandler* oraz używa obiektu klasy *MessageReader*. Implementuje ona metodę *run(),* która w nowo wystartowanym wątku łączy się z aplikacją serwera, oraz odczytuje wiadomość powitalną. Wiadomość ta wypisywana jest na strumień oraz zapisywana jest do zmiennej statycznej klasy *MainActivity* – s_*welcomeMessage*.

Kolejną omawianą klasą jest klasa bazowa klasy *ServerConnectionHandler* to jest klasa *CommonHandler*. Klasa ta implementuje interfejs *Runnable* biblioteki Android. Na rys.6.15 przedstawiono implementację metody *connectToServer*() klasy *CommonHandler*. Metoda ta jest metodą chronioną to znaczy dostęp do niej mają jedynie klasy pochodne, jej zadaniem jest nawiązanie połączenia z wskazanym adresem.

```
package pl.kobak.rafal.dicommobile.pl.kobak.rafal.utilities;
import android.util.Log;
import pl.kobak.rafal.dicommobile.MainActivity;

/**
 * Created by Rafal Kobak on 2016-05-13.
 */
public class ServerConnectionHandler extends CommonHandler
{
    public ServerConnectionHandler()
    {
        super();
    }

    @Override
    public void run()
    {
        super.connectToServer();
        MessageReader l_msgReader = new MessageReader();
        Message l_msg = l_msgReader.readMessage();

        printWelcomeMessage(l_msg);
        MainActivity.s_welcomeMessage = l_msg.getUsefulPayload();
    }

    private void printWelcomeMessage(Message p_msg)
    {
        Log.d(LABEL, "Received message: ");
        Log.d(LABEL, "msgId: " + p_msg.msgId.name());
        Log.d(LABEL, "numOfMsgInFileTransfer: " + p_msg.numOfMsgInFileTransfer);
        Log.d(LABEL, "bytesInPayload: " + p_msg.bytesInPayload);
        Log.d(LABEL, "payload: " + p_msg.getUsefulPayload());
    }
}
```

Rys.6.12. Definicja klasy *ServerConnectionHandler*.

W metodzie mamy blok try – catch, który mówi o tym, że kod zawarty w bloku try może rzucić tak zwany wyjątek łapany w poszczególnych blokach catch. Pierwszą operacją w metodzie jest konwersja adresu IP z typu *String* na typ *InetAdress* wykorzystywany w klasie *Socket*() języka JAVA. Następnie za pomocą klasy *Socket*() tworzone jest gniazdo sieciowe i ustanawiane jest połączenie. Poniższe bloki catch wychwytują konkretne typy wyjątków, które mogą być rzucone przez kod w bloku try.

```
protected void connectToServer()
{
    try
    {
        InetAddress l_serverAddress = InetAddress.getByName(m_serverIp);
        MainActivity.s_socket = new Socket(l_serverAddress, m_serverPort);

        Log.d(LABEL, "Connection with " + m_serverIp +
                " in port " + m_serverPort +
                " successfully established.");
    }
    catch (UnknownHostException e)
    {
        Log.d(LABEL + " exception", "Exception occurred: Unknown host: " + m_serverIp + "!");
        Log.d(LABEL + " exception", e.getMessage());
    }
    catch (IOException e)
    {
        Log.d(LABEL + "_exception", "Exception occurred: No I/O!");
        Log.d(LABEL + " exception", e.getMessage());
    }
    catch (Exception e)
    {
        Log.d(LABEL + "_exception", "Unknown exception occurred!");
        Log.d(LABEL + "_exception", e.getMessage());
    }
}
```

Rys.6.13. Definicja metody *connectToServer*() kalsy *ServerConnectionHandler*.

Ostatnią klasą używaną w kodzie źródłowym omawianego ekranu jest klasa *MessageReader*. Klasa ta posiada jedną metodę publiczną *readMessage*() przedstawioną na rys.6.14. Metoda ta odczytuje wiadomość a następnie zwraca ją do użytkownika klasy. W metodzie tej ponownie znajduje się blok try – catch, co wskazuje na to, że kod wewnątrz bloku try może rzucać wyjątek. Początkowo tworzona jest tablica bajtów o rozmiarze odpowiadającym rozmiarze wiadomości. Następnie z otwartego gniazda sieciowego pobierany jest strumień wejściowy odczytywana jest wiadomość. Odczyt następuje partiami pole po polu za pomocą prywatnych metod klasy *Message*.

```
public Message readMessage()
{
    try
    {
        byte[] l receivedRawMsg = new byte[MSG SIZE];
        InputStream l inputStream = MainActivity.s socket.getInputStream();
        int l_readBytes = l_inputStream.read(l_receivedRawMsg, 0, l_receivedRawMsg.length);

        Log.d(LABEL, "Amount of read bytes: " +  Integer.toString(l_readBytes));

        fillMessageId(l receivedRawMsg);
        fillNumOfMsgInFileTransfer(l receivedRawMsg);
        fillBytesInPayload(l_receivedRawMsg);
        fillPayload(l_receivedRawMsg);
    }
    catch (IOException e)
    {
        Log.d(LABEL + "_exception", e.getMessage());
        Log.d(LABEL + "_exception", "Exception occurred in read message: No I/O!");
    }

    return m message;
}
```

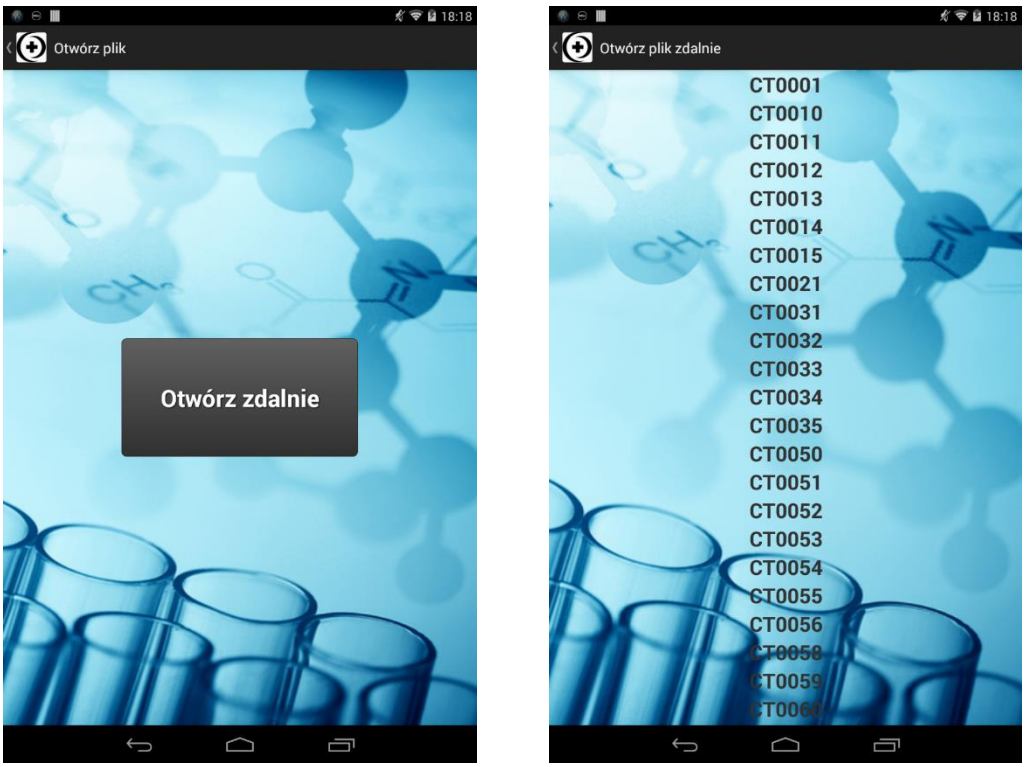
Rys.6.14. Definicja metody *readMessage*() klasy *MessageReader*.

6.3.3. Ekrany „Otwórz plik” oraz „Otwórz plik zdalnie”.

Ekran „Otwórz plik” zawiera jeden duży centralnie umieszczony przycisk „Otwórz zdalnie”. Dodatkowo na ekranie znajduje się menu umożliwiające powrót do ekranu macierzystego. Ekranem pierwotnym ekranu „Otwórz plik” jest ekran główny. Ekran „Otwórz plik” przedstawiony został na rys.6.16. W przyszłości możliwe jest dodanie funkcji lokalnego odczytu plików. Mógłoby to polegać na trzymaniu jednokrotnie sparsowanych plików w pamięci telefonu i w razie potrzeby podglądania wyników w trybie offline. Po naciśnięciu centralnie umieszczonego przycisku otwierany jest nowy ekran to jest „Otwórz plik zdalnie”. Ekran ten jest podekranem ekranu „Otwórz plik”. Jest to pierwszy omawiany ekran, który nie jest bezpośrednim podekranem ekranu głównego. Funkcja będąca uchwytem do przycisku – *onClick_openFileRemotely()* przedstawiona została na rys.6.15.

```
public void onClick openFileRemotely(View p view)
{
    Intent l_openFileRemotely = new Intent(this, OpenFileRemotely.class);
    startActivity(l_openFileRemotely);
}
```

Rys.6.15. Ciało publicznej metody *onClick_openFileRemotely()*.



Rys.6.16. Ekrany "Otwórz plik" oraz "Otwróz plik zdalnie".

W pierwszej linii ciała metody tworzony jest obiekt typu *Intent* inicjalizowany klasą *OpenFileRemotely* a więc klasą źródłową ekranu „Otwórz plik zdalnie". Następnie aktywność jest uruchamiana za pomocą funkcji *startActivity()* przyjmującej, jako parametr uprzednio utworzony obiekt typu *Intent*.

Ekran „Otwórz plik zdalnie" wypełniony jest widokiem listy – *ListView*. Na liście tej zgromadzone są udostępnione przez serwer pliki DICOM. Wybór konkretnego pliku do parsowania odbywa się za pomocą dotknięcia przez użytkownika jego nazwy na ekranie tabletu. Klasą źródłową omawianego ekranu jest *OpenFileRemotely*, której publiczną metodę *onCreate*() przedstawiono na rys.6.17.

W omawianej metodzie początkowo ustawiana jest zawartość ekranu jeszcze bez listy plików, według pliku *activity_open_file_remotely.xml.* Następnie tworzony jest wątek odpowiadający za uzyskanie listy plików dostępnych na serwerze. Klasa *ServerFileListHandler* realizująca to zadanie zostanie omówiona w dalszej części rozdziału. Kolejną częścią po uzyskaniu listy plików jest podział jej na odpowiednie podłańcuchy znaków, gdzie każdy łańcuch stanowi nazwę jednego pliku. Tablica takich łańcuchów przechowywana jest w zmiennej *l_receivedFileList*.

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_open_file_remotely);

    Thread l_connectionThread = new Thread(new ServerFileListHandler());
    l_connectionThread.start();

    while(l_connectionThread.isAlive()) {}

    String [] l_receivedFileList = MainActivity.s_fileList.split("\\r?\\n");
    LinearLayout l_linearLayout = new LinearLayout(this);
    setContentView(l_linearLayout);
    l_linearLayout.setOrientation(LinearLayout.VERTICAL);

    for( int i = 0; i < l_receivedFileList.length; i++ )
    {
        if(l_receivedFileList[i].endsWith(".txt") || l_receivedFileList[i].endsWith(".png"))
            continue;

        TextView l_textView = new TextView(this);
        l_textView.setText(l_receivedFileList[i]);
        l_textView.setClickable(true);
        l_linearLayout.addView(l_textView);
        l_textView.setOnClickListener(new OnClickListenerCustom());
        l_textView.setTextSize(getResources().getDimension(R.dimen.text_on_list));
        l_textView.setGravity(Gravity.CENTER);
        l_textView.setTypeface(Typeface.DEFAULT_BOLD);
    }
    getWindow().getDecorView().setBackgroundResource(R.drawable.ogolny2);
}
```

Rys.6.17. Ciało publicznej metody *onCreate().*

Następującą operacją jest utworzenie w programie widoku liniowego, który będzie zbudowany w ten sposób, że każdy pojedynczy wiersz będzie nazwą pojedynczego pliku. Iterując w pętli po otrzymanej liście plików program odrzuca pliki kończące się na .png oraz .txt (rezultaty parsowania wcześniejszych plików) i dodaje kolejne wiersze w postaci pól tekstowych do widoku liniowego. Każde pole ma zdefiniowane w ten sam sposób takie parametry jak reakcja na kliknięcie, rozmiar tekstu, jego pogrubienie oraz wyśrodkowanie. W przypadku kliknięcia na wybrane pole tekstowe, czyli element listy, sterowanie jest przekazywane do uchwytu klasy *OnClickListenerCustom*(), którym jest metoda *onClick*().Implementacja metody *onClick*() klasy *OnClickListenerCustom*() przedstawiona została na rys.6.18.

```
@Override
public void onClick(View p_view)
{
    TextView l_textView = (TextView) p_view;
    MainActivity.s_chosenFileName = l_textView.getText().toString();

    Intent l_openFile = new Intent(p_view.getContext(), FileWindow.class);
    p_view.getContext().startActivity(l_openFile);

    Thread l_parseDicomFile_connectionThread = new Thread(new ServerParseDicomFileHandler());
    l_parseDicomFile_connectionThread.start();

    while(l_parseDicomFile_connectionThread.isAlive()) {}

    Thread l_requestParsedFiles_connectionThread
            = new Thread(new ServerFileSentHandler(MainActivity.s_chosenFileName + ".png"));
    l_requestParsedFiles_connectionThread.start();
    while(l_requestParsedFiles_connectionThread.isAlive()) {}
}
```

Rys.6.18. Implementacja metody *onClick()* klasy *OnClickListenerCustom.*

Po wywołaniu metody *onClick*() pobierana jest wartość tekstu z kliknięty pola tekstowego i zapisywana jest w statycznej zmiennej klasy *MainActivity*. Następnie tworzony jest nowy podekran o nazwie „Okno pliku” w podobny sposób jak omówiony ekran „Otwórz plik zdalnie”. Tworzony jest więc podekran trzeciego rzędu. Kolejną operacją w metodzie jest wysłanie żądania parsowania wybranego przez użytkownika pliku realizowane przez klasę *ServerParseDicomFileHandler* działającą na osobnym wątku. Po otrzymaniu pozytywnej odpowiedzi od serwera wysyłane jest żądanie przesłania wynikowego pliku z obrazem realizowane przez klasę *ServerFileSentHandler*.

Metoda *run*() klasy *ServerParseDicomFileHandler* przedstawiona została na rys.6.19. W metodzie nawiazywane jest połączenie z serwerem, następnie tworzony jest obiekt *MessageReader* odczytujący wiadomość powitalną i wysyłana jest wiadomość SERVER_PARSE_DICOM_FILE_REQ będąca żądaniem parsowania oraz odbierana odpowiedź z rezultatem, czyli SERVER_PARSE_DICOM_FILE_RESP. Wpomniane operacje odbywają się w metodach pomocniczych owawianej klasy to jest metodach *sendServerParseDicomFileReq*() oraz *receiveServerSendDicomFileResponse*() przedstawionych na rys.6.20.

```
@Override
public void run()
{
    super.connectToServer();
    MessageReader l_msgReader = new MessageReader();
    l_msgReader.readMessage();

    sendServerParseDicomFileReq();
    receiveServerSendDicomFileResponse();
}
```

Rys.6.19. Implementacja metody *run()* klasy *ServerParseDicomFileHandler*.

```
private void sendServerParseDicomFileReq()
{
    Message l_msg = buildServerSendDicomFileReq();
    MessageSender l_msgSender = new MessageSender();
    l_msgSender.send(l_msg);
}

private Message buildServerSendDicomFileReq()
{
    Message l_msg = new Message();
    String l_payload = "./moduleTest/plikiTestyAndroid/" + MainActivity.s_chosenFileName;
    l_msg.msgId = EMessageId.SERVER_PARSE_DICOM_FILE_REQ;
    l_msg.numOfMsgInFileTransfer = 0;
    l_msg.bytesInPayload = l_payload.length();
    l_msg.payloadWrite = l_payload.toCharArray();

    return l_msg;
}

private void receiveServerSendDicomFileResponse()
{
    MessageReader l_msgReader = new MessageReader();
    Message l_msg = l_msgReader.readMessage();

    printReceivedMessage(l_msg);
}
```

Rys.6.20. Implementacja metod pomocniczych klasy *ServerParseDicomFileHandler*.

W nieco bardziej skomplikowany sposób odbywa się przesłanie pliku przez klasę *ServerFileSentHandler*. Metoda *run()* wspomnianej klasy przedstawiona została na rys.6.21. Po nawiązaniu połączenia i odebraniu wiadomości powitalnej następuje wysłanie żądania przesłania pliku z obrazem. Nazwa pliku określona jest w zmiennej składowej *m_fileName*. W wiadomości oprócz nazwy pliku zawarta jest cała ścieżka do lokalizacji gdzie znajdują się pliki będące wynikiem parsowania to jest *./moduleTest/plikiTestyAndroid/*.

```
@Override
public void run()
{
    super.connectToServer();
    MessageReader l_msgReader = new MessageReader();
    l_msgReader.readMessage();

    sendServerSendFileReq();
    receiveServerSendFileResponse();

    receiveSendFileInd();
}

private void sendServerSendFileReq()
{
    Message l_msg = buildServerSendFileReq();
    MessageSender l_msgSender = new MessageSender();
    l_msgSender.send(l_msg);
}

private Message buildServerSendFileReq()
{
    Message l_msg = new Message();
    String l_payload = "./moduleTest/plikiTestyAndroid/" + m_fileName;
    l_msg.msgId = EMessageId.SERVER_SEND_FILE_REQ;
    l_msg.numOfMsgInFileTransfer = 0;
    l_msg.bytesInPayload = l_payload.length();
    l_msg.payloadWrite = l_payload.toCharArray();

    return l_msg;
}
```

Rys.6.21. Ciała metod *run()*, *sendServerFileReq()* oraz *buildServerSendFileReq()*.

W otrzymanej od serwera odpowiedzi zawarta jest informacja w ilu wiadomościach CLIENT_SEND_FILE_IND zostanie przesłany sam plik z obrazem – *numOfMsgInFileTransfer*. Informacja ta jest zapisywana w celu wykorzystania przy transmisji samego pliku.

Implementacja metod służących do obierania pliku przedstawiona została na rys.6.22. Początkowo otwierany jest strumień do zapisu strumienia. Strumień ten posłuży do zapisywania danych do nowo utworzonego pliku w pamięci wewnętrznej tabletu. Następnie w metodzie *receiveFile*() w pętli for otrzymywane są kolejne wiadomości CLIENT_SEND_FILE_IND, z których odczytywane jest pole danych i zapisywane kolejno bajt po bajcie w nowo otwartym strumieniu.

```
private void receiveSendFileInd()
{
    BufferedOutputStream l bufferedOutputStream = getBufferedOutputStream(m fileName);

    try
    {
        receiveFile(l bufferedOutputStream);
        l bufferedOutputStream.flush();
        l_bufferedOutputStream.close();
    }
    catch (IOException e)
    {
        Log.d(LABEL + "_exception", e.getMessage());
        e.printStackTrace();
    }
}

private void receiveFile(BufferedOutputStream p_bufferedOutputStream)
        throws IOException
{
    for (int j = 0; j < m_numOfMsgInFileTransfer; j++)
    {
        MessageReader l msgReader = new MessageReader();
        Message l msg = l msgReader.readMessage();

        for (int i = 0; i < l_msg.bytesInPayload; i++)
        {
            p bufferedOutputStream.write(l msg.payloadRead[i]);
        }
    }
}
```

Rys.6.22. Ciała metod *receiveSendFileInd()* oraz *receiveFile().*

Ostatnią klasą warta omówienia jest klasa *ServerFileListHandler* zajmująca się wysyłaniem żądania przesłania listy plików i odebraniem odpowiedzi. Metoda *run*() wspomnianej klasy przedstawiona została na rys.6.23. Podobnie jak w poprzednio omawianych klasach tak i tutaj początkowo nawiązywane jest połączenie z serwerem, odbierana jest wiadomość powitalna, następnie budowane i wysyłane jest żądanie przesłania listy plików SERVER_SEND_FILE_LIST_REQ. Kolejno odbierana jest odpowiedź, w której polu danych znajduję się lista plików dostępnych na serwerze. Lista ta zapisywana jest w zmiennej statycznej *s_fileList*.

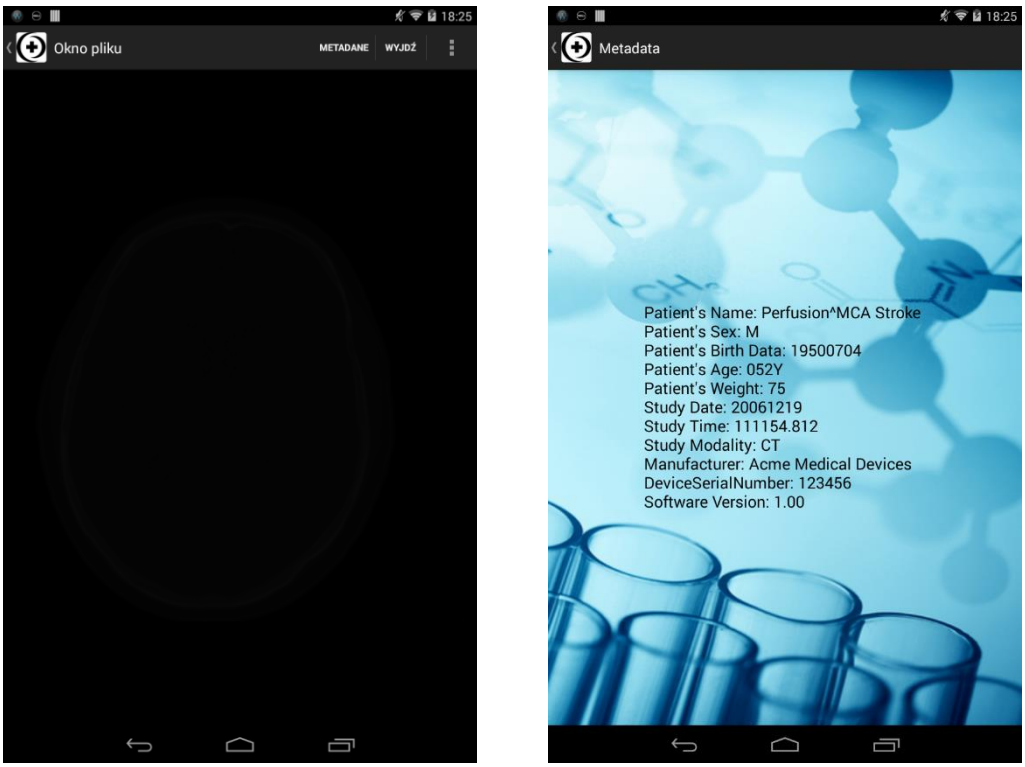
```
@Override
public void run()
{
    Log.d(LABEL, "Adress:" + MainActivity.s_ipAddress);
    Log.d(LABEL, "PORT:" + MainActivity.s_portNumber);
    super.connectToServer();
    MessageReader l_msgReader = new MessageReader();
    l msgReader.readMessage();

    sendServerSendFileListRequest();
    receiveServerSendFileListResponse();
}
```

Rys.6.23. Ciało metody *run()* klasy *ServerFileListHandler.*

6.3.4. Ekrany „Okno pliku” oraz „Metadane”.

Ekran „Okno pliku” służy do prezentacji obrazu będącego rezultatem parsowania wybranego pliku DICOM. Z kolei ekran „Metadane” służy do prezentacji danych tekstowych będących wynikiem parsowania tego samego pliku. Ekran „Okno pliku” jest ekranem macierzystym ekranu „Metadane”.

Ekran „Okno pliku” poza menu wypełniony jest w całości elementem *ImageView*, na którym prezentowany jest wynikowy obraz. Plikiem XML z danymi ekranu jest plik *activity_file_window.xml* natomiast klasą z kodem źródłowym jest klasa *FileWindow*. Dla ekranu „Metadane” są to kolejno plik *activity_metadata.xml* oraz klasa *Metadata*. Oba ekrany przedstawione zostały na rys.6.24.



Rys.6.24. Ekrany „Okno pliku” oraz „Metadane”.

Na rys.6.25 przedstawiono implementację metody *onCreate*() klasy *FileWindow*. Metoda ta jest wywoływana automatycznie po uruchomieniu ekranu. Pierwszą operacją w metodzie jest ustawienie widoku według zdefiniowanego dla ekranu pliku xml.

Następnie pobierany jest widok obrazu *ImageView* za pomocą globalnego identyfikatora. Kolejnym krokiem jest określenie pełnej ścieżki do pliku w lokalizacji lokalnej tabletu. Ścieżka ta zapisywana jest w zmiennej *l_path*. Dalej za pomocą obiektu klasy *BitMap* wczytywany jest do programu obraz o lokalizacji określonej w *l_path*.

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_file_window);

    ImageView l_imageView = (ImageView) this.findViewById(R.id.ImageView);
    String l_rootPath = Environment.getExternalStorageDirectory().getAbsolutePath();
    String l_path = l_rootPath + File.separator + "mojePliki"
                  + File.separator + MainActivity.s_chosenFileName + ".png";
    Log.d(LABEL, l_path);

    Bitmap l_bmp = BitmapFactory.decodeFile(l_path);
    if(l_bmp == null)
    {
        Log.d(LABEL,
              "Nie udalo sie wczytac pliku: " + MainActivity.s_chosenFileName + ".png !");
    }
    l_imageView.setImageBitmap(l_bmp);
}
```

Rys.6.25. Ciało metody *onCreate()* klasy *FileWindow.*

W przypadku niepowodzenia logowany jest odpowiedni komunikat, w przypadku powodzenia zawartość elementu *ImageView* ustawiana jest na wczytany obraz. W ten sposób na ekranie pojawia się obraz będący rezultatem parsowania wybranego przez użytkownika pliku DICOM.

Na rys.6.26 przedstawiono implementację metody *onCreate*() klasy *Metadata*. Metoda ta podobnie jak poprzednio omawiana ustawia najpierw widok zgodnie z definicjami w pliku xml. Następnie tworzony jest watek, na którym uruchamiana jest klasa *ServerFileSentHandler*. Tym razem zadaniem klasy *ServerFileSentHandler* jest uzyskanie pliku tekstowego będącego rezultatem parsowania wybranego pliku DICOM. Podobnie jak w przypadku pliku z obrazem wysyłane jest żądanie przesłania wybranego pliku tesktowego, a w odpowiedzi przychodzi ilość plików składająca się na przesyłany plik. Dalej wysyłany jest w częściach sam plik tekstowy.

Kolejną operacją w funkcji jest uzyskanie lokalnej ścieżki do pliku i otwarcie pliku. Dalej konieczne jest przetworzenie uzyskanych danych do postaci umożliwiającej wypisanie na ekran. Wyżej opisane operację mogą prowadzić do rzucania wyjątków toteż wymagane jest zastasowanie bloków try – catch. Wyodrębniony z pliku tekst jest finalnie ustawiony w elemencie *TextView* ekranu – 6.24.

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_metadata);

    startRequestParsedFilesConnectionThread();

    TextView l_textView = (TextView) this.findViewById(R.id.textView);
    String l_pathToFile = getPathToFile();
    Log.d(LABEL, l_pathToFile);

    String l_textFromFile = getTextFromFile(l_pathToFile);
    l_textView.setText(l_textFromFile);
}

private void startRequestParsedFilesConnectionThread()
{
    Thread l_requestParsedFiles_connectionThread
            = new Thread(new ServerFileSentHandler(MainActivity.s_chosenFileName + ".txt"));

    l_requestParsedFiles_connectionThread.start();
    while(l_requestParsedFiles_connectionThread.isAlive()) {}
}

private String getPathToFile()
{
    String l_rootPath = Environment.getExternalStorageDirectory().getAbsolutePath();
    String l_path =  l_rootPath + File.separator
                                + "mojePliki"
                                + File.separator
                                + MainActivity.s_chosenFileName
                                + ".txt";
    return l_path;
}

private String getTextFromFile(String p_filePath)
{
    BufferedReader l_bufferReader = null;
    try
    {
        l_bufferReader = new BufferedReader(new FileReader(p_filePath));
    }
    catch (FileNotFoundException e)
    {
        Log.d(LABEL + "_exception", "Blad przy otwieraniu pliku!");
        Log.d(LABEL + "_exception", e.getMessage());
        e.printStackTrace();
    }

    StringBuilder l_total = new StringBuilder();
    String l_line = new String();
    try
    {
        while((l_line = l_bufferReader.readLine()) != null)
        {
            l_total.append("\n");
            l_total.append(l_line);
        }
    }
    catch (IOException e)
    {
        Log.d(LABEL + "_exception", "Blad przy odczycie");
        Log.d(LABEL + "_exception", e.getMessage());
        e.printStackTrace();
    }
    return l_total.toString();
}
```

Rys.6.28. Ciało metody *onCreate()* oraz metod pomocniczych klasy *Metadata.*

7. Podsumowanie.

Celem nieniejszej pracy było stworzenie systemu wizualizacji, danych medycznych DICOM z możliwością dostępu zdalnego. Udało się zrealizować system wizualizacji, który – zgodnie z założeniami – złożony jest z dwóch aplikacji: serwera oraz klienta. Dostęp aplikacji klienta do programu serwera ma charakter zdalny – Wi-Fi. Wizualizacja odbywa się z wykorzystaniem biblioteki DCMTK. Zgodnie z założeniami aplikacja serwera bierze na siebie ciężar operacji parsowania pliku DICOM oraz świadczenia usług bazodanowych. Program ten umożliwia obsługę wielu klientów jednocześnie. Aplikacja kliencka jest możliwie lekka, przystosowana do działania na średniej wydajności tablecie.

W docelowym miejscu działania systemu, program serwera powinien zostać zainstalowany na jednostce stacjonarnej bądź laptopie o stosunkowo wysokiej wydajności. Po uzupełnieniu bazy danych o pliki DICOM, będące rezultatem badań medycznych, system umożliwi prezentację obrazów czy danych tekstowych z wykorzystaniem tabletu w dowolnym miejscu w zasięgu sieci Wi-Fi. System taki po dopracowaniu i dostosowaniu mógłby być zainstalowany w szpitalu. Zlikwidowałoby to potrzebę transportu pacjentów do pomieszczeń ze stacjami roboczymi i ułatwiło przygotowanie pacjentów do zabiegu.

Dużym problemem podczas realizacji pracy okazała się komunikacja pomiędzy aplikacjami napisanymi w różnych językach programowania, dodatkowo działających na różnych systemach operacyjnych. Rozwiązaniem okazało się zdefiniowanie wspólnego interfejsu w postaci systemu wiadomości o określonym rozmiarze, które są wymieniane przez aplikacje w określonej kolejności.

Aplikacja serwera ze względu na swoją architekturę otwarta jest na dodawanie nowych funkcjonalności. Funkcjonalności takie można dodawać poprzez dopisywanie kolejnych klas jednostek obsługujących *Handler* oraz wiadomości będących wyzwalaczami dla tych jednostek. Każda jednostka obsługująca jest niezależna, nie ma, więc potrzeby zastanawiania się nad interakcjami pomiędzy jednostkami obsługującymi w przypadku rozszerzania aplikacji. Aplikację serwera można rozszerzyć o jednostkę obsługującą, która zajmuje się renderowaniem poszczególnych obrazów wyekstraktowanych z pliku DICOM. Dodatkowym rozszerzeniem byłoby stworzenie jednostki obsługującej łączącej się z szpitalnych serwerem PACS. Co więcej należałoby dodać system uwierzytelniania klientów oraz szyfrowanie transmisji tak, aby dostęp do danych pacjentów był możliwy jedynie przez autoryzowane do tego osoby. Z punktu widzenia, jakości kodu w aplikacji serwera można by wyodrębnić wspólny interfejs dla klas jednostek obsługujących.

Aplikację kliencką można rozszerzać poprzez dodawanie kolejnych ekranów w aplikacji. Przykładowym rozszerzeniem byłaby na przykład możliwość otwarcia plików, (które zostały wcześniej przetworzone przez aplikację serwera) lokalnie. Umożliwiłoby to dostęp do obrazów w trybie offline. Dodatkowo protokół komunikacyjny można rozszerzyć o Bluetooth. Innym rozszerzeniem byłoby wyświetlanie wielu obrazów z pliku DICOM pochodzących z jednego badania w postaci animacji poklatkowej.

8. 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