

WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI



PRACA DYPLOMOWA

STUDIA II^o

Temat pracy: **MOBILNY SYSTEM ZARZĄDZANIA I STEROWANIA BEZPILOTOWYM STATKIEM LATAJĄCYM**

INFORMATYKA

.....
(kierunek studiów)

SYSTEMY INFORMATYCZNE

.....
(specjalność)

Dyplomant:

Norbert WASZKOWIAK

Promotor:

dr inż. Michał DYK

Warszawa 2022

OŚWIADCZENIE

*Wyrażam zgodę / ~~nie wyrażam zgody~~ **
na udostępnianie mojej pracy przez Archiwum WAT

Dnia

.....

(podpis)

* *Niepotrzebne skreślić*

Spis treści

Wstęp	5
Rozdział I. Prezentacja zagadnienia bezpilotowych statków latających oraz koncepcji ich wykorzystania	6
I.1. Definicja BSP	6
I.2. Historia BSP	6
I.3. Technologia i producenci BSP	11
I.4. Zastosowania BSP	13
I.5. Dostosowywanie BSP do wymagań użytkownika	16
Rozdział II. Przegląd i prezentacja technologii mobilnych z uwzględnieniem aspektów tworzenia aplikacji i komunikacji M2M	19
II.1. Komunikacja M2M	19
II.2. Technologie komunikacji M2M	20
II.3. Komunikacja bezprzewodowa w dronach konsumenckich	23
II.4. Kontrolowanie BSP za pomocą API dostarczanego od producenta	29
Rozdział III. Projekt mobilnego systemu zarządzania i sterowania BSP	30
III.1. Wymagania funkcjonalne	30
III.2. Wymagania pozafunkcjonalne	30
III.3. Stos technologiczny	30
III.4. Wysokopoziomowy diagram systemu	31
III.5. Diagram komponentów	32
III.6. Diagramy klas	34
III.7. Wykorzystane urządzenia	41
Rozdział IV. Implementacja systemu	43
IV.1. Wykonywanie komend	43
IV.2. Obsługa MQTT	46
IV.3. Kontrolowanie BSP	48
IV.4. Interfejs użytkownika	53
Rozdział V. Testy systemu oraz prezentacja użycia na wybranym case study	54
V.1. Testy na platformie Andorid	54
V.2. Testy jednostkowe	55
V.3. Testy na środowisku uruchomieniowym	57
V.4. Testy end-to-end	60

V.5. Testy manualne	62
Podsumowanie	63
Bibliografia	64
Spis rysunków	67
Spis tabel	67
Załączniki	68

Wstep

Rozdział I. Prezentacja zagadnienia bezpilotowych statków latających oraz koncepcji ich wykorzystania

I.1. Definicja BSP

W nomenklaturze związanej z domeną bezpilotowych statków latających można znaleźć wiele tożsamyh terminów na określanie bezpilotowych statków latających, są to m.in.:

- Bezzałogowy statek powietrzny, BSP (ang. *unmanned aerial vehicle*, UAV);
- Bezzałogowy system powietrzny (ang. *unmanned aerial system*, UAS);
- Samolot zdalnie sterowany (ang. *remotely piloted aircraft*, RPA);
- Dron (ang. *drone*),

Każdy z tych terminów kładzie nacisk na inną cechę, ale wszystkie nadal odnoszą się do jednego obiektu i będą w tej pracy używane zamiennie.

Amerykański pisarz zajmujący się zagadnieniami systemów bezzałogowych i technologii obronnych, Kelsey Artheon na łamach czasopisma *Popular Science* definiuje to pojęcie następująco: "dron oznacza każdy bezzałogowy zdalnie sterowany pojazd latający, bez względu na to, czy jest to malutki, sterowany radiem helikopter-zabawka, czy też ważący 14,5 tony Global Hawk, wart 104 mln dolarów. Jeżeli coś lata i jest sterowane przez pilota z ziemi, to pasuje do potocznej definicji drona".¹ Biorąc to pod uwagę, można zdefiniować następujące warunki do zakwalifikowania obiektu jako bezzałogowy statek powietrzny:

- **bezpilotowość** - na swoim pokładzie nie posiada pilota;
 - **dwukierunkowość** - musi mieć możliwość powrotu/wylądowania (Jest to podstawowa cecha odróżniająca drony od pocisków manewrujących);
 - **sterowalność** - możliwość zmiany kierunku lotu w trakcie jego wykonywania.
- [1][2]

I.2. Historia BSP

Po przedstawieniu definicji BSP można przystąpić do przedstawienia historii całej tej domeny, wraz ze wskazaniem jej początku w poprawny sposób.

I.2.1. Błędnie klasyfikowane obiekty

Po zdefiniowaniu czym jest dron, można się zastanowić, co było pierwszym elementem spełniającym tę definicję. Autor tej pracy uważa, że kluczowym elementem umożliwiającym zakwalifikowanie obiektu jako dron jest możliwość zmiany trajektorii lotu w trakcie jego działania. W literaturze często wskazywane są dwa obiekty jako prekursorzy dronów, tzn. gołąb Archytasa z Tarentu i balony zawierające ładunki wybuchowe

¹ A. Kelsey, *Flying Robots 101: Everthing You Need to Know about Drones*, Popular Science[2]

wykorzystane w konflikcie między Austrią i Wenecją w 1849 r. Pierwszy rzekomy prekursor nie umożliwia sterowania obiektem po jego wystartowaniu, więc tym samym nie jest to zgodne z przytoczonymi definicjami. Ten wynalazek można uznać za pierwszą rakietę lub robota, ale nie drona. Drugi przykład, czyli balony na gorące powietrze, również nie mogą zostać uznane za bezzałogowy statek powietrzny z tego samego powodu. Jako ciekawostkę można dodać, że pomysł Austriaków zakończył się niepowodzeniem, ponieważ wiatr zwał balony na ich własne pozycje.[1].



Rys. 1. Latający gołąb Archytasa z Tarentu

Źródło: <https://input.niezalezna.pl/259fef5fd.jpg>

I.2.2. Pierwszy pełnoprawny dron

Podczas I wojny światowej podjęto liczne próby skonstruowania bezpilotowych statków latających, ale żaden z nich nie został ukończony przed skończeniem wojny. Przykładowo *Kettering Bug*, był w stanie dolecieć na odpowiednią odległość, ale jego sterowanie polegało na wyliczeniu przez operatora dokładną liczbę obrotów silnika. Mając to na uwadze, takiemu samolotowi bliżej do torpedy niż do drona.

W 1931 r. Królewskie Siły Powietrzne (ang. Royal Air Force) na podstawie samolotu szkolnego *De Havilland DH-60T "Tiger Moth"* opracowywały pierwszy bezpilotowy statek powietrzny *DH-82B "Queen Bee"*. Samolot ten, sterowany przez pilota za pomocą fal radiowych, miał służyć jako ruchomy cel do ćwiczeń dla obsługi dział przeciwlotniczych. Jego oficjalna prezentacja została jednak przerwana, ponieważ ówczesne systemy obrony powietrznej były tak mało skuteczne, że strzelającym skończyła się amunicja, zanim zestrzelili oni bezpilotowy samolot. Obiekt ten też spełnia wszystkie wymagania określone wcześniej przez autora, więc uznaje on go za pierwszego drona.

Równolegle w tym samym okresie, a konkretnie w 1935 r. powstał identyczny samolot dla amerykańskich odbiorców, *Radioplane OQ-2*. Powstał on jako pierwotnie jako bezzałogowiec, a nie przez modyfikacje tak jak dron brytyjski, więc jego budowa bardziej odstawała od klasycznych samolotów.[10][1]

I.2.3. Pierwsze drony rozpoznawcze

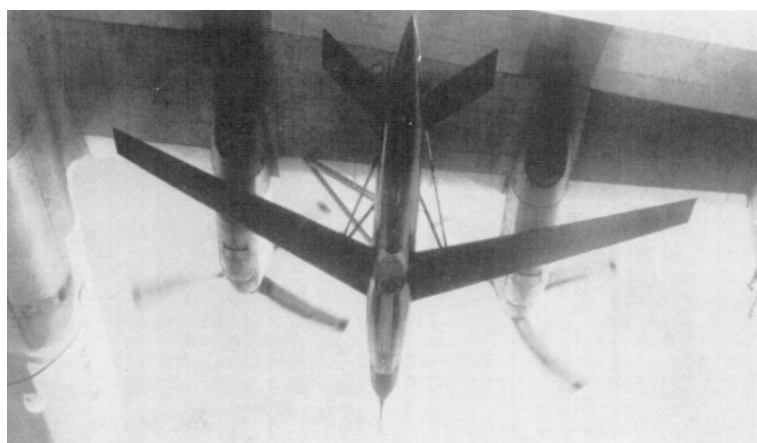
W bezpośrednim okresie po II wojnie światowej USA kontynuowało prace nad dronami, poprzez firmę *Ryan Aeronautical Company* i ich serii dronów *Firebee*, produkowanych



Rys. 2. De Havilland Queen Bee i premier Wielkiej Brytanii Winston Churchill

Źródło: <https://www.iwm.org.uk/collections/item/object/205195356>

od 1951 r. Efektem tej serii był m.in. opracowany w 1962 r. *Ryan Model 147 Lightning Bug*. Był to dron rozpoznawczy, napędzany był za pomocą silnika rakietowego. Nie posiadał on wyposażenia do lądowania i startowania z ziemi, tak więc odbywała się to za pomocą spadochronu, w który był wyposażony, i jego przechwyceniu w locie przez helikopter. Sam start odbywał się z pokładu samolotu. Tak samo, jak pociski rakietowe, dron ten był umieszczany pod skrzydłem samolotu.



Rys. 3. Ryan Model 147 Lightning Bug umieszczony pod skrzydłem samolotu transportowego

Źródło: <https://en.wikipedia.org/wiki/File:>

Model_147_RPV_pictured_in_flight_under_wing_pylon_of_a_carrier_aircraft.png

I.2.4. Ikona wśród BSP

Zdecydowanie do najpopularniejszego BSP na świecie należy zaliczyć, produkowanego przez amerykańskiego producenta, *General Atomics MQ-1 Predator*. Pierwsze jego wersje nie posiadały na swoim pokładzie żadnych pocisków, ponieważ rząd USA nie byli pewni czy jest to zgodne z obowiązującym układem dotyczącym całkowitej likwidacji pocisków rakietowych średniego zasięgu (ang. *Intermediate-range Nuclear Forces (INF) Treaty*). Jednak wydarzenia z 11 września 2001 r. były impulsem do podjęcia decyzji o uzbrojeniu Predatorów w pociski rakietowe i skierowania ich do akcji. Umożliwiło to prowadzenie operacji militarnych bez ponoszenia strat w żołnierzach. Drony te były wykorzystywane w trakcie konfliktów w Afganistanie, Iraku czy Pakistanie. Na przestrzeni lat 2009-2021 Zjednoczona Ameryka stała się światowym liderem w używaniu dronów bojowych.[1][21]



Rys. 4. MQ-1 Predator, wyposażony w rakiety AGM-114 Hellfire

Źródło: https://en.wikipedia.org/wiki/File:MQ-1_Predator,_armed_with_AGM-114_Hellfire_missiles.jpg

I.2.5. Drony cywilne

Trudno zaprzeczyć stwierdzeniu, że wojna przyczynia się do szybszego rozwoju, bo to właśnie rozwiązania opracowane dla armii przenoszone są często do życia codziennego. Było tak z herbatą ekspresową, jak jest i teraz z bezzałogowymi statkami powietrznymi. Zmieniła się tylko ich rola, za ich pomocą nie prowadzi się działań wojennych, a nagrywa sceny do filmów, prowadzi transmisje skoków narciarskich z ciekawszej perspektywy i ratuje ludzi zagubionych w górach. Okolice obecnego roku można wskazywać jako okres największego zainteresowania tą technologią na rynku cywilnym. Początku tego okresu można próbować określać na 2013 r., czyli datę premiery pierwszej wersji, prawdopodobnie najpopularniejszej serii dronów *Phantom* od obecnie najpopularniejszego producenta dronów konsumenckich *DJI*.

I.2.6. Konflikt na Ukrainie

W kontekście bezzałogowych statków powietrznych nie można pominąć aktualnego konfliktu zbrojnego na Ukrainie. Należy go rozpatrywać w dwóch aspektach: przewagi, która armia ukraińska uzyskuje dzięki tureckim dronom *Bayraktar TB2* i wykorzystaniu dronów konsumenckich od ludności cywilnej do przeprowadzenia rozpoznania powietrznego.

Rząd ukraiński zwrócił się z prośbą do swoich obywateli o przekazanie swoich dronów na potrzeby armii. Są one wykorzystywane do bezpiecznego prowadzenia rozpoznania przez wojska ukraińskie. Dostarczają one obraz na żywo, wraz ze swoimi współrzędnymi geograficznymi. Pozwala to budować przewagę informacyjną na polu bitwy, a ten konflikt szczególnie uświadomił, jak ważna jest dzisiaj informacja na polu bitwy.[12]

Czytając artykuły poświęcone dronom *Bayraktar TB2* w kontekście konfliktu, można odnieść wrażenie, jakby było to jedyny element budujący ich siłę. Autor nie może się z tym zgodzić, ale nie da się zaprzeczyć, że ich rola jest znacząca. Szczególnie po obejrzeniu licznych nagrań dostępnych w internecie z działań tych samolotów na wojnie. Głównym celem tej maszyny jest jednak prowadzenie rozpoznania, ale mogą być one wyposażone w cztery pociski kierowane o zasięgu 8 km. Sam dron jest jedną z tańszych opcji na rynku, bo jego cena waha się między 2-6 mln dolarów, a drony z najwyższej półki sięgają 100 mln dolarów. Rozpiętość skrzydeł tego drona to 12 metrów, a długość to 6,5 metra, co przekłada się na możliwość szybowania przez 27 godzin lub przelecenia 150 km. W dodatku wzbija się na pułap 8200 m i rozwija prędkość do 220 km/h, a wszystko to za sprawą silnika *Rotax 912 iS* o mocy 100 koni mechanicznych.[13][14]



Rys. 5. Bayraktar TB2

Źródło: https://www.instalki.pl/images/newsy/03-2022/Bayraktar_TB2_ukraina.jpg

I.3. Technologia i producenci BSP

Wymagania stawiane dronom na rynku cywilnym i militarnym znacznie się od siebie różnią, tak samo, jak konstrukcje które je spełniają. Największą różnicą jest właśnie rozmiar. Militarne BSP do realizacji swoich zadań muszą pokonywać dużo większe dystanse, przewożąc przy tym ciężkie wyposażenie. Te dwa środowiska można też uznać za hermetyczne względem siebie, rzadko następuje przejście z jedno w drugie. Większość producentów dronów wojskowych wcześniej produkowała inny sprzęt wojskowy, a cywilnych kamery czy elektryczne szybowce.

I.3.1. Shenzhen DJI Sciences and Technologies Ltd.

Shenzhen DJI Sciences and Technologies Ltd., znany powszechnie pod nazwą handlową DJI, jest obecnie największym producentem dronów konsumenckich. Z siedzibą w chińskiej "dolinie krzemowej" Shenzhen. Firma została założona w 2006 r., a swój pierwszy sukces odniosła w 2013 r. kiedy wpuściła na rynek pierwszy model serii dronów *Phantom*. Był to BSP przeznaczony dla początkujących operatorów, a na tle konkurencji wyróżniała go łatwość obsługi. W kolejnych latach firma kontynuowała swój rozwój, a w 2015 r. wraz z wypuszczeniem trzeciej wersji jego wersji stała się ona największym producentem na świecie. W 2016 r. firma ta posiadała 50% udziałów w światowym rynku, a rok później już 72%. W 2020 r. było około 74%, podczas gdy żadna inna firma w tym samym czasie nie posiadała więcej niż 5% udziału w światowym rynku.

W 2020 r. BSP firmy DJI były wykorzystane przez Chiny do przypominania ludziom o obowiązku noszenia maseczek na twarzy w celu ograniczenia rozprzestrzeniania się wirusa COVID-19. Z tego samego powodu w takich krajach jak Maroko czy Arabia Saudyjska drony mierzyły temperatury przemieszczającej się populacji w terenach silnie zurbanizowanych.[9][11]

I.3.2. Yuneec International

Yuneec International to drugi co do wielkości producent dronów konsumenckich, pomimo że w rynku światowym posiada zaledwie 5% udziałów. Firma ta została założona w 1999 r. i pierwotnie zajmowała się produkcją samolotów elektrycznych i szybowców. Obecnie jej siedziba znajduje się w Jiangsu w Chinach. Do swojej oferty wprowadziła pierwszego drona w 2015 r. Był on przeznaczony do fotografii.

W 2014 r. firma została członkiem założycielem *Dronecode*, organizacji non-profit prowadzonej przez *Linux Foundation*, której celem jest dostarczanie otwartego oprogramowania do dronów opartego na jądrze systemu Linux. Firma *Intel Corporation* w sierpniu 2015 r. zainwestowała 60 mln dolarów w Yuneec w zamian za 15% udziałów. Celem inwestycji była wspólna realizacja przyszłych projektów. W tym samym miesiącu Yuneec wprowadził na rynek drona *Breeze*, zdolnego do rejestrowania filmów i zdjęć w rozdzielczości UltraHD 4K.

Również w 2015 r. firma nawiązała współpracę z Ocean Alliance, organizacją zajmującą się ochroną wielorybów. Celem było stworzenie bezpieczniejszego sposobu na zbieranie danych o stanie zdrowia wielorybów. Organizacja Ocean Alliance, zamiast

korzystać, tak jak dotychczas z rzutek biopsyjnych zaczęła używać do tego celów specjalnie wyposażonych dronów Yeneec.[7]

I.3.3. Baykar

W tym zestawieniu nie mogło zabraknąć producenta najpopularniejszego drona militarnego w kontekście aktualnego konfliktu zbrojnego na Ukrainie, czyli: *Baykar*. Jest to prywatna firma Turecka specjalizująca się w obszarach sztucznej inteligencji, BSP i systemach *Command And Control* (pl. dowodzenie i kontrola). Została ona założona w 1984 r. przez Özdemir Bayraktar i pierwotnie dostarczała części samochodowe takie jak pompy, silniki i inne. W latach dwutysięcznych firma zainteresowała się obszarem BSP, czego efektem było wyprodukowanie *Bayraktar Mini UAV*. Był to pierwszy dron w całości wyprodukowany z kapitału krajowego Turcji i w 2007 r. znalazł się na wyposażeniu Tureckich Sił Zbrojnych. Rozpoczęcie prac badawczo-rozwojowych przyczyniło się do produkcji pionierskich i zaawansowanych systemów. Portfolio firmy obejmuje również latający samochód, nazwany *Cezeri*, który w trakcie testów w Stambule we wrześniu 2020 r. Wzniósł się na wysokość 10 m.[22]



Rys. 6. Dron osobowy Cezarei w trakcie testów w Stambule

Źródło: <https://www.savunmahaber.com/en/wp-content/uploads/2020/09/CEZERI-UCAN-ARABA-26.jpg>

I.3.4. General Atomics

General Dynamics Corporation to amerykańska korporacja założona w 1952 r., z siedzibą w Reston, w stanie Wirginia. Do 1990 r. dostarczała on czołgi, rakiety, pociągi, łodzie podwodne, okręty wojenne, myśliwce i elektronikę dla wszystkich rodzajów

wojsk. Na początku lat 90tych sprzedała całe swoje portfolio, z wyjątkiem działalności związanej z pojazdami wojskowymi i okrętami podwodnymi.

Korporacja ta jest właścicielem *General Atomics Aeronautical Systems*, która zajmuje się produkcją systemów radiowych i bezzałogowych statków powietrznych, w tym rewolucyjnego kiedyś drona wojskowego *Predator*. Seria tych dronów jest nadal rozwijana, a poszczególne konstrukcje są do siebie bardzo zbliżone.[28][27]

I.4. Zastosowania BSP

Bezpilotowe statki latające znalazły szereg zastosowań, pierwotnie były wykorzystywane głównie w obszarze militarnym, dopiero później dostrzeżono w nich potencjał również w środowisku cywilnym. Ponieważ kontekst militarny został już dość szczegółowo przedstawiony, w poniższym tekście przyłożono większą uwagę do ich cywilnych zastosowań.

I.4.1. Militarne zastosowania BSP

Bezzałogowe statki powietrzne w kontekście militarnym można dokonać podziału na następujące kategorie:

- **bojowe** - przenoszące i używające środki bojowe/ środki rażenia, np. *Bayraktar TB2*;
- **amunicja krążąca** - umożliwiające wykrywanie, rozpoznanie oraz atak na wyznaczony cel poprzez autodestrukcyjne, np. *WB Electronics Warmate*;
- **operacyjno-rozpoznawcze** - realizujące rozpoznawanie oraz śledzenie obiektu/celu, a także monitorowanie i kontrole obszaru zainteresowania, np. granic lub strefy przybrzeżnej. Przykładem takiego drona jest *Lockheed Martin RQ-170 Sentinel*;
- **wsparcia**: umożliwiające ewakuacje lub dostawę amunicji, wyposażenia, środków medycznych i żywności do wysuniętych stanowisk wojsk własnych, np. *Kaman KARGO UAV* [26]

I.4.2. Cywilne zastosowania BSP

Wymienienie wszystkich cywilnych rozwiązań jest trudne, ponieważ rynek ten znajduje co chwilę kolejne zastosowania dronów. Poniżej przedstawiono parę wyselekcjonowanych interesujących rozwiązań.

Transport medyczny

W czerwcu 2022 r. Polska Agencja Żeglugi Powietrznej wydała zgodę na wykonywanie regularnych długodystansowych lotów BSP. Realizować będzie je firma transportowa na rzecz systemu opieki zdrowotnej. Połączenie obejmie dwie trasy Warszawa-Pułtusk i Warszawa-Sochaczew. Przewidywana częstotliwość lotów to ok. 7 w tę i z powrotem

w ciągu dnia. Obie te trasy mają długość 60 km i odbywają się na wysokości 100 m. Sam samolot posiada systemy wizyjne, które będą korygowały lot w przypadku wystąpienia przeszkód na jego trasie przelotu. Takie połączenie zapewni szybki transport np. narządów do przeszczepu co może przyczynić się do uratowania komuś życia. [29]



Rys. 7. Dron *Farada G1*, za pomocą którego będzie odbywał się transport medyczny w okolicach Warszawy

Źródło: <https://www.pansa.pl/wp-content/uploads/2022/02/IMG-20220213-WA0001-1024x577.jpg>

Ratownictwo

Drony powietrzne pomagają również w ratowaniu ludzkiego życia, szczególnie w górach. W styczniu 2022 r. ze względu na trudne warunki atmosferyczne ratownicy TOPR nie mogli udzielić pomocy dwóm turystą, którzy nie byli w stanie zejść z góry. Za pomocą drona dostarczono im koce i ogrzewacze, co pozwoliło im przetrwać noc. Kolejnego dnia, gdy pogoda uległa poprawie ratownicy dotarli do poszkodowanych i ich sprowadzili w bezpieczny sposób.[30]

Również na dalekim zachodzie można znaleźć przykłady ratowania życia z użyciem BSP, a konkretnie w Północnej Kalifornii. Właśnie tam, w lesie, zgubił się młody myśliwy. Służby za pomocą drona zlokalizowali jego lokalizację, a następnie wysłali tam strażników, którzy z jego pomocą wyprowadzili zagubionego.[32]

Są cztery powody, dla którego BSP dobrze odnajdują się w ratownictwie.

- **Czas reakcji** - są opcją bardzo szybkiego reagowania, czas potrzebny na przygotowanie do startu jest minimalny;
- **Szybkie przeszukiwania** - mogą przeszukać trudny teren w dużo szybszym czasie niż człowiek pieszo;
- **Komunikacja** - mogą komunikować się z poszkodowanymi za pomocą głośnika i mikrofonu;

- **Lokalizacja** - są w stanie przekazywać na żywo do operatora swoją aktualną lokalizację, co w przypadku ratowania i poszukiwania osób może znacznie minimalizować czas poszukiwania i ewakuacji. [31]

Kinematografia i produkcja filmowa

Jednym z pierwszych filmów, które przywoływane jako przykład dobrego wykorzystania dronów jest *Skyfall* z 2012 r., a konkretnie scena pościgu motorem przez Jamesa Bonda złoczyńców po dachach w Istambule. BSP dały kinematografii wyjątkową przewagę nad tradycyjnymi metodami filmowania. Mają większy zasięg niż żuraw i są też bardziej zwinne niż helikopter. Reżyserzy dzięki temu mogą wykonywać bardziej ryzykowne, prawdziwie akrobatyczne ujęcia, które gdyby nie drony musiałyby być wytworzone na komputerze. Nie można zapomnieć, że BSP są przede wszystkim tańsze w zakupie i utrzymaniu niż np. helikoptery.[34]

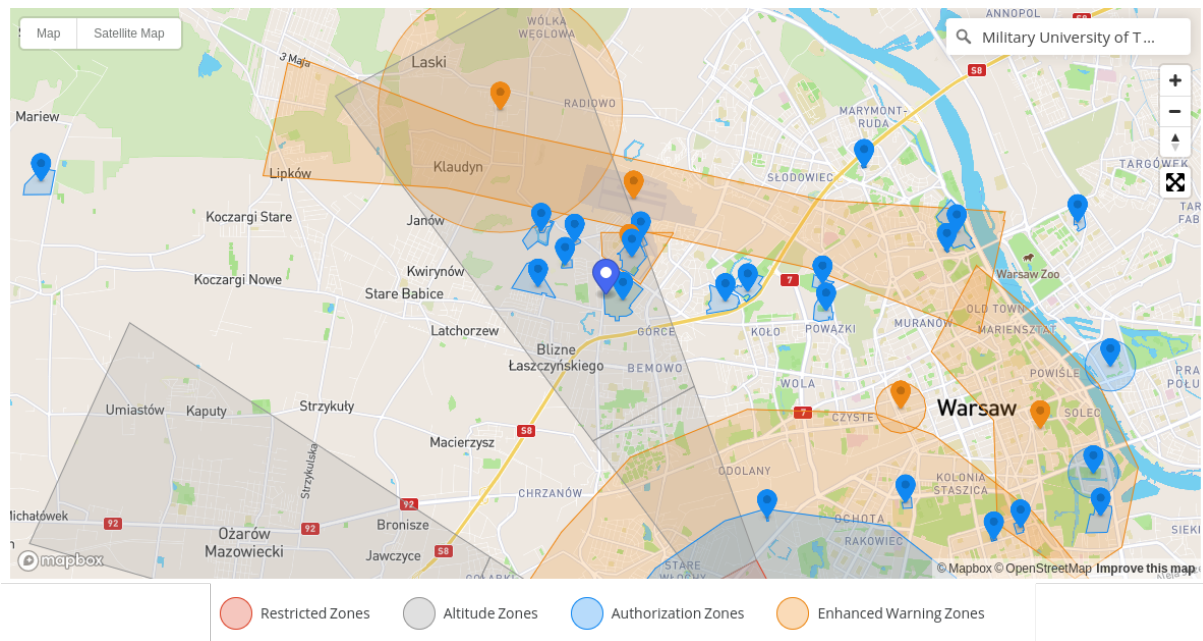
Przykładem gdzie wcześniej nie było możliwe nagrywanie ujęć filmowych, są erupcje wulkanów. Takie loty wiązały się z dużym niebezpieczeństwem, a drony są dużo tańsze i nie posiadają na swoim pokładzie załogi, tak więc operatorzy mogą pokusić się o takie ryzyko. Takie nagranie wykonał znany youtuber Joey Helms, który postanowił uwiecznić z bliska rzekę lawy wyrzucaną przez wulkan Fagradalsfjall na Islandii. W trakcie nagrywania erupcja lewy w pewnym momencie sięgnęła jego statku. Youtuber bezpowrotnie stracił swój statek. Ujęcie, które zostało zachowane, jest bardzo emocjonujące, więc prawdopodobnie było to warte swojej ceny.[33]

Transport towarów

W 2013 r. Jeff Bezos ogłosił koncepcję, która miała zrewolucjonizować transport towarów. Autonomiczne drony powietrzne miały dostarczać paczki Amazonu w 30 minut pod drzwi odbiorcy. Mogły one zaoferować konsumentom dostawę żywności, leków czy innych lżejszych przedmiotów, bez spalania paliw kopalnianych i czekania. Niestety jednemu z najbogatszych ludzi na świecie nie udało się tego osiągnąć. Tak samo, jak firmie *Zipline*, która miała dostarczać leki w Rwandzie i projektowi Google *Wing*, który miał zapewniać burrito głodnym studentom. Na świecie istnieje duża presja na wprowadzenie takiego rozwiązania. W 2021 r. Amazon zwolnił większość pracowników odpowiedzialnych za rozwój wspomnianego projektu, uzasadnił to błędem w zarządzaniu i panującym chaosem. W tym samym roku DHL również ogłosił zakończenie swojego identycznego projektu. Było to 8 lat po tym, gdy ich dron pierwszy raz wbił się w powietrze. Koszty transportu towaru między centrum dystrybucji a klientem końcowym to 40% całkowitego kosztu, a konsumenci oczekują szybkiej taniej i ekologicznej dostawy. Wszystkie te potrzeby mogłyby zaspokoić dostawy dronami.

Jest parę czynników, które mogły wpłynąć na wstrzymanie masowych dostaw towarów dronami. Głównym z nich jest prawdopodobnie ograniczenie stref lotów BSP. Osoby zarządzające państwami i miastami na bieżąco wprowadzają regulacje w tej dynamicznie rozwijającej się domenie. Wprowadzone zostały m.in. strefy wyłączone dla dronów, które mają zapewnić bezpieczeństwo wokół miejsc wymagających specjalnej ochrony. Są to najczęściej okolice lotnisk i jednostek wojskowych. Takie strefy mocno

ograniczają dostawy towarów dronami.[36]



Rys. 8. Strefy wyłączone dla dronów w okolicach Warszawy

Źródło: <https://www.dji.com/pl/flysafe/geo-map>

Mimo wszystko taki transport jest jednak nadal możliwy, ale nie na masową skalę. W Sosnowcu od 2021 r. realizowany jest projekt, w którym drony będą świadczyły usługi transportu m.in. pilnych przesyłek medycznych. Odbiór i nadawanie będzie odbywało się w stacjach dokujących, które będą automatycznie ładowały i zdejmowały ładunek z drona. Wykorzystanie takich punktów umożliwia ustawienie stałej trasy lotów, co zdejmuje dużą część odpowiedzialności z uczestników projektu, dlatego też w przyszłości można spodziewać się większej ilości tego typu rozwiązań.[37]

I.5. Dostosowywanie BSP do wymagań użytkownika

Na rynku znajduje się duża ilość dostępnych komponentów, z których można łatwo zbudować własne bezzałogowe statki powietrzne. Najwięcej odpowiedzialności z konstruktora zdejmują gotowe kontrolery lotu, np. *pixhawk*, na którym można zainstalować otwarte oprogramowanie *ArduPilot*. Pozwala to zachować dużą ilość zasobów, bo cała logika lotu jest praktycznie *out-of-the-box*. Przykładowy zestaw elementów potrzebnych do zbudowania własnego drona to::

- rama,
- silnik,
- elektroniczny kontroler prędkości,

- śmigła,
- załącza,
- rozdzielnia zasilająca,
- baterie,
- monitoring baterii,
- mata montażowa,
- kontroler,
- odbiornik RC,
- kamera,
- karta pamięci SD.[35]



Rys. 9. Kontroler lotu Pixhawk

Źródło: https://ardupilot.org/plane/_images/Pixhawk_with_legend.jpg

Budowa własnego drona jest jak najbardziej możliwa, bez wymagania specjalistycznej wiedzy, ale nadal to potrzeba na to dużo ilości zasobów, szczególnie czasu, dlatego warto przyrzeć się gotowym rozwiązaniom dostarczonym przez producentów i to jak bardzo można je dostosowywać. Firma DJI udostępnia do swoich produktów SDK, czyli bibliotekę, za pomocą której można zaprogramować działanie naszego drona. Dodatkowo producent dla wybranych statków przygotował szereg rozszerzeń, które pomogą dostosować wyposażenie do naszych wymagań. Przykładowo dla *Mavic 2 Enterprise Advanced* dostępne są:

- **głośnik** - który umożliwia komunikację z drona, np. w czasie sytuacji alarmowych do ludności znajdującej się na ziemi
- **dodatkowe oświetlenie** - w przypadku wystąpienia niekorzystnych warunków atmosferycznych lub lotów w nocy
- **moduł RTK** - który umożliwia osiągnięcie dokładności lokalizacji na poziomie jednego centymetra

Dodatkowo dla innych modeli dostępne są wyspecjalizowane kamery np. do skanu obiektów w 3D czy podglądu w podczerwieni.

Rozdział II. Przegląd i prezentacja technologii mobilnych z uwzględnieniem aspektów tworzenia aplikacji i komunikacji M2M

W tym zdefiniowano cel i wymagania stawiane technologią M2M. Następnie przedstawiono przykładowe technologie ze szczególnym uwzględnieniem komunikacji pomiędzy bezzałogowym statkiem powietrznym a kontrolerem. Rozdział zakończono analizą interfejsu API umożliwiającego kontrolowanie drona.

II.1. Komunikacja M2M

Komunikacja M2M (machine-to-machine) to kategoria technologii, która umożliwia wymianę informacji pomiędzy urządzeniami w sieci bez jakiegokolwiek ingerencji ludzi. Obiekty pracujące w tej sieci charakteryzują się mniejszą lub większą autonomią. Wspierana jest ona często szeroko pojętą sztuczną inteligencją, ze szczególnym uwzględnieniem technik uczenia maszynowego. Ta komunikacja jest podstawą istnienia IoT.[4]

II.1.1. Cel

Głównym celem M2M jest autonomiczna komunikacja pomiędzy maszynami, a jej obecnie najczęstszym zastosowaniem jest przenoszenie danych z sensorów do sieci. Obecnie operatorzy coraz częściej rozbudowują swoją infrastrukturę o węzły zgodnie z tą kategorią technologii. Dzięki temu koszty utrzymania całego systemu są znacznie zredukowane. Składają się na niego następujące elementy:

- łącze do przesyłu danych, np. WiFi, GSM
- sensory, np. czujnik temperatury, kamera
- oprogramowanie, które automatyzuje procesy komunikacyjne, np. przeszukiwanie ścieżki routingu

Celem telemetrii jest automatyczny pomiar wielkości fizycznej przez odpowiednie sensory. Wartość pomiaru jest przesyłana do miejsca, zwykle odległego, w którym jest dalej przetwarzana. Na początku do tego celu były wykorzystywane linie telefoniczne, a następnie radiowe. Rozwój technologii, a w tym łączności bezprzewodowej sprawił, że poszerzyła się rola wykorzystywania telemetrii w nauce, inżynierii i produkcji. Dzisiaj jest ona używana także w życiu codziennym, w jednostkach grzewczych, miernikach elektrycznych i wszelkich urządzeniach podłączonych do internetu. Jej rozwój jest ściśle powiązany z komunikacją M2M.[4]

II.1.2. Wymagania

Według Europejskiego Instytutu Norm Telekomunikacyjnych (ETSI) komunikacja M2M musi spełniać następujące wymagania:

- **Skalowalność:** w miarę dołączania kolejnych urządzeń do systemu system nadal musi funkcjonować;
- **Anonimowość:** w związku z wymaganiami prawnymi, na każde żądanie system musi umożliwiać ukrywanie tożsamości urządzenia;
- **Logowanie:** ważne wydarzenia w systemie, takie jak: pojawienie się błędnych informacji czy nieudane próby instalacji, muszą być zarejestrowane, a rejestry te muszą być dostępne na żądanie;
- **Zasady komunikacji między aplikacjami:** aplikacje w systemie powinny mieć możliwość komunikowania się. W szczególności bramki i urządzenia końcowe komunikujące się za pomocą technologii SMS czy Ethernet powinny komunikować się za pomocą połączenia P2P (peer-to-peer);
- **Metody dystrybucji:** w ramach systemu powinny być dostarczane metody dystrybucji takie jak: *unicast*, *multicast*, *broadcast* i *anycast*, a wszędzie gdzie to możliwe metoda *broadcast* powinna być zastąpiona za pomocą *multicast*, tak aby zminimalizować obciążenie sieci;
- **Harmonogram przesyłania komunikatów:** dostęp do sieci powinien być kontrolowany, tak samo, jak harmonogram przesyłania komunikatów. Sam system powinien również uwzględniać obciążenia aplikacji M2M w harmonogramie przesyłania wiadomości;
- **Wybór ścieżek komunikacyjnych:** ścieżki w systemie powinny zapewniać optymalizację bazującą na: awariach transmisji, kosztu i opóźnieniach, w momencie, gdy istnieją inne ścieżki do punktu docelowego. [4]

II.2. Technologie komunikacji M2M

Poprawnym stwierdzeniem będzie, że obecnie znajdujemy się w epoce połączonych ze sobą obiektów. IoT (Internet of Things) zdobywa aktualnie coraz więcej uwagi nie mały w każdej dziedzinie, a szczególnie w takich jak biznes, elektronika konsumencka, przemysł czy transport. Niemal każdy obiekt elektryczny w dzisiejszym świecie jest ze sobą połączony w ten czy inny sposób. Siedząc w biurze, za pomocą dostarczanych aplikacji, możemy kontrolować drzwi, bramę garażową, czajnik elektryczny czy rolety okienne w naszym domu. Z kolei w mieście kontrolujemy kamery i oświetlenie, a wszystko to z odległych lokacji. IoT odgrywa w tym ważną rolę, ponieważ to ono umożliwia łączenie przeróżnych obiektów, za pomocą sieci połączeń i wymianę danych między nimi.[6]

II.2.1. Ogólna klasyfikacja technologii M2M

Poniżej przedstawiono ogólne porównanie technologii komunikacyjnych M2M.

	Local Area Network Komunikacja krótko dystansowa	Low Power Wide Area Internet Of Things	Cellular Network Tradycyjne M2M
Użycie	40%	45%	15%
Zalety	- Dobrze ugruntowana norma - W budynkach	- Niskie zużycie energii - Niskie koszty - Pozycjonowanie	- Istniejące pokrycie znacznego obszaru - Duża prędkość transmisji
Wady	- Wysokie zużycie energii elektrycznej - Duży koszt sieci i zależności	- Niska prędkość transmisji - Wschodzący standard	- Wysoki koszt posiadania - Mała autonomia
Technologia	Bluetooth, WiFi	LoRa	GSM, 3G, 4G, 5G

Tab. 1. Porównanie rodzajów technologii M2M.[6]

Źródło: Badania własne.

II.2.2. LoRa

Komunikacja w aplikacjach IoT jest dzisiaj wykonywana w przeróżnych technologiach, a każda z nich ma swoje zalety, funkcje, a przez to też przeznaczenie. Żadna z tych technologii nie może pokryć całego zapotrzebowania świata IoT, ponieważ wszystkie one posiadają cechy, które czynią je odpowiednie dla postawionego konkretnego zadania.

LoRa (Long Range) to nowa technologia połączeń bezprzewodowych w świecie IoT. Ostatnio znacznie ewoluowała i zyskała szczególną popularność w urządzeniach z ograniczoną pojemnością elektryczną, umożliwiając systemom wbudowanym przesyłanie małej ilości danych na dużych dystansach w krótkich interwałach czasowych.

WiFi to najpopularniejsza technologia komunikacji bezprzewodowej, która jest już rozwijana przez wiele lat. W kontekście IoT wykorzystywana jest przede wszystkim do komunikacji na dużych odległościach. Na krótkie dystanse lepiej pasują do tego takie protokoły jak Bluetooth czy ZigBee. We wszystkich z nich największą wadą jest duże zużycie energii elektrycznej. Technologia LoRa zapewnia bezpieczne, mobilne dwukierunkowe połączenie o niskim koszcie elektrycznym. Wykorzystywane jest ono w IoT, szczególnie w domenie smart city, czy nawet ogólnej komunikacji M2M. LoRa zalicza się do LPWA (Low Power Wide Area), czyli rodzaju bezprzewodowej rozległej sieci telekomunikacyjnej, stworzonej w celu umożliwienia komunikacji na duże odległości przy niskiej przepływności i niskim poborze energii. [5] W tego typu komunikacji wyróżnia się LoRa ze względu na jej:

- długodystansowość;
- dwukierunkowość;

- wysoką pojemność węzłów w sieci;
- długość życia na baterii;
- odporność interfejsów;
- bezpieczeństwo i efektywność sieci. [6]

Cechy

Technologie tą wyróżniają następujące cechy:

- Pojedyncza bramka może pokryć obszar aż 100km^2 ;
- Oferuje ona podwójne szyfrowanie AES;
- Bazuje na technologii CSS (widmo rozproszone Chirp), które umożliwia śledzenie obiektów i jest odporne na zanikanie sygnałów;
- Topologia gwiazdy eliminuje zanikanie danych przez urządzenia pośrednie, co przyczynia się do zmniejszenia poboru mocy. [6]

Ograniczenie przepustowości

W sieci LoRa wszystkie klasy ramek wymagają potwierdzenia. Wiąże się z tym, że po każdym potwierdzeniu ramki przez urządzenie końcowe w dowolnym oknie czasowym następuje okres wyłączenia, w celu zachowanie zgodności z przepisami dotyczącymi cyklu pracy. W związku z tym, aby uniknąć wyczerpania limitu pojemności przez sieć i urządzenia końcowe, muszą one ograniczyć liczbę potwierdzeń. Również w podsieciach LoRa po przesłaniu danych następuje okres wyłączenia, w którym na danym kanale nie są wysyłane żadne dane. Te dwa okresy, tzn. okres wysyłania danych i wstrzymania transmisji stanowi cykl pracy. Cały ten mechanizm przyczynia się do ograniczenia przepustowości sieci.[6]

II.2.3. Narowband IoT

Wraz z rozwojem świata IoT zyskała również technologia Narowband IoT (NB-IoT). Wykorzystywana jest ona przede wszystkim przy komunikacji komórkowej dla zdalnych pomiarów w całej Europie. Jest to technologia dostępu radiowego. Używa ona ponownie komponentów stworzonych przez jej poprzednika LTE, aby umożliwić jej działanie na licencjonowanej częstotliwości. Może ona również działać w trybie autonomicznym. Tak jak sama nazwa wskazuje, cały system działa w wąskim spektrum częstotliwości, bo tylko w 200kHz, co wprowadza elastyczność zastosowań dzięki minimalnym wymaganiom częstotliwości, w porównaniu do jej poprzednika LTE. Cała szerokość 200kHz została podzielona na kanały po 3.75 kHz lub 15 kHz, co umożliwia połączenie w bardzo wysoką prędkość nadawania, a także daleki zasięg połączenia. [8]

II.2.4. NB-IoT vs Lora

Tab. 2. Porównanie technologii LoRa i NB-IoT [3]

Parametr	LoRa	NB-IoT
Pasma	125 kHz	180 kHz
Pokrycie	165 dB	164 dB
Żywotność baterii	15+ lat	10+ lat
Maksymalne natężenie elektryczne	32 mA	120 mA
Spoczynkowe natężenie elektryczne	1 μ A	5 μ A
Przepustowość	50 Kbps	60 Kbps
Opóźnienie	Zależne od klasy urządzenia	10 s
Bezpieczeństwo	AES 128 bit	3GPP (128 to 256 bit)
Geolokalizacja	Tak (TDOA)	Tak (In 3GPP Rel 14)
Jakość/cena	Wysoka	Średnia

Źródło: Badania własne.

Zarówno LoRa, jak i NB-IoT należą do wspomnianej wcześniej technologii LPWAN. Podstawowe różnice pomiędzy tymi dwoma technologiami można dostrzec w zużyciu baterii, prędkości transmisji i opóźnieniach.

II.3. Komunikacja bezprzewodowa w dronach konsumenckich

Przeglądając katalog największego producenta dronów konsumenckich DJI, można wyróżnić tylko 3 technologie komunikacji bezprzewodowej: wzmacniona WiFi (ang. enhanced WiFi), Lightbridge, OcuSync.

II.3.1. WiFi

WiFi nie zostało wprowadzone ściśle do komunikacji bezprzewodowej statków powietrznych, ale odnajduje się w tym całkiem dobrze. Jest ona wykorzystywana głównie w bardziej budżetowych wersjach dronów, ze względu na możliwość skorzystania przez producenta z posiadanej przez użytkownika infrastruktury (smartfonów), czy niskiej ceny komponentów.

Przykładowo dron *DJI Tello*, który jest najtańszą opcją dostępną od producenta DJI, przeznaczoną głównie do nauki latania i programowania, również przez najmłodszych pasjonatów. Nie posiada on w zestawie dedykowanego kontrolera, ponieważ odbywa się ono za pomocą aplikacji na smartfona, która łączy się z dronem za pomocą

WiFi tak jak do punktu dostępowego z internetem. Zasięg takiego połączenia według producenta to 100 m.[16]



Rys. 10. DJI Tello

Źródło: <https://store.dji.com>

W swojej ofercie DJI ma również dostępnego drona *DJI Mini SE*, który również korzysta z technologii WiFi, ale w swoim wyposażeniu posiada dedykowany do niego kontroler. Taka konfiguracja pozwala na uzyskanie zasięgu do 2 km. [15]

WiFi jest także bardzo podatne na wszelkie zakłócenia, wynikające z ukształtowania terenu czy zaszumienia sieci pochodzącego z istniejących sieci domowych. Wyprodukowanie drona w tej technologii jest najtańszą dostępną opcją, która umożliwia transmisję obrazu, jednak należy pamiętać, aby nie stawiać jej przy tym za dużych wymagań. Stanowi ono dobry punkt startowy w komunikacji bezprzewodowej bezzałogowych statków powietrznych.

II.3.2. Lightbridge

Lightbridge to technologia od DJI, która doczekała się jej dwóch wydań. Pierwszych wzmianek o niej można doszukiwać się w 2014 r., a drugiego wydania już w 2015 roku. Jest ona zbliżona do technologii WiFi, przede wszystkim transmisja ta odbywa się na tej samej częstotliwości 2,4GHz.

Była ona kierowana głównie do dronów z wyższego pułapu cenowego, dlatego że jej produkcja była bardzo kosztowna, a koszt wynikał z tego, że producent opracował to rozwiązanie na swoim autorskim układzie scalonym i oprogramowaniu. Umożliwiło to osiągnąć duże lepsze wyniki niż transmisja po WiFi. Zasięg lotu według producenta to odległość do 5 km.

Obecnie Lightbridge nie jest już rozwijany, a producent skupił się na jego następniku, technologii OcuSync. [17][18]

II.3.3. OcuSync

OcuSync został po raz pierwszy zademonstrowany przez producenta wraz z wydaniem drona *Mavic Mini Pro*. Pierwsze wydanie tej technologii pozwalało na transmisję do 7 km na częstotliwości 2,4 GHz. Obraz mógł być przesyłany w rozdzielczości 720p i 1080p. Jakość fullHD była dostępna tylko na krótszych odległościach. Na większych dystansach gdy dostępna prędkość transmisji spadała, dron przechodził automatycznie na transmisję w 720 p. Opóźnienie było rzędu 160-170ms. A największą cechą wyróżniającą tę technologię była możliwość podłączenia jednocześnie dwóch kontrolerów i do 4 urządzeń odbiorczych.

Kolejnym krokiem było wydanie wersji oznaczonej jako OcuSync 1.5, w której dodano transmisję również na częstotliwości 5 Ghz. Zmniejszono także opóźnienia w transmisji. Dodatkowo technologia umożliwiała automatyczną zmianę kanałów komunikacyjnych w trakcie lotu na te najmniej obciążone. W pierwszej wersji kanał transmisji można było wybrać tylko przed startem bezzałogowego statku powietrznego.[24]



Rys. 11. Pierwsza wersja gogli do FPV od DJI

Źródło: https://u.cyfrowe.pl/600x0/2/7/2_732250420.png

Wraz z wydaniem nowej wersji zaprezentowano *gogle DJI* przeznaczone do transmisji obrazu w trybie FPV (ang. first person view, widok pierwszo-osobowy) i również *OcuSync Aircraft System*, czyli zintegrowanego systemu umożliwiającego sterowanie i transmisji obrazu z wykorzystaniem tej technologii w dronach i pojazdach DIY.

Producent w trakcie swojej historii doprowadził do pewnych nieścisłości. Pomimo że dron *Phantom 4 pro v 2.0* korzystał teoretycznie z najnowszej wersji OcuSync, nie posiadał on możliwości zmiany kanałów transmisji w trakcie lotu, a opóźnienie zależało też od tego, czy korzystano z kontrolera dołączonego do zestawu, czy jego droższej, lepiej wyposażonej wersji: *DJI RC Plus*.

Wersja 2.0 wprowadziła dalsze ulepszenia, m.in. kontrolowanie dronów na jeszcze większe dystanse i z jeszcze mniejszymi opóźnieniami, a także kompatybilność wsteczną po aktualizacji oprogramowania.



Rys. 12. Dji OcuSync Air Unit

Źródło: <https://store.dji.com>



Rys. 13. DJI Phantom v2.0

Źródło: <https://store.dji.com>

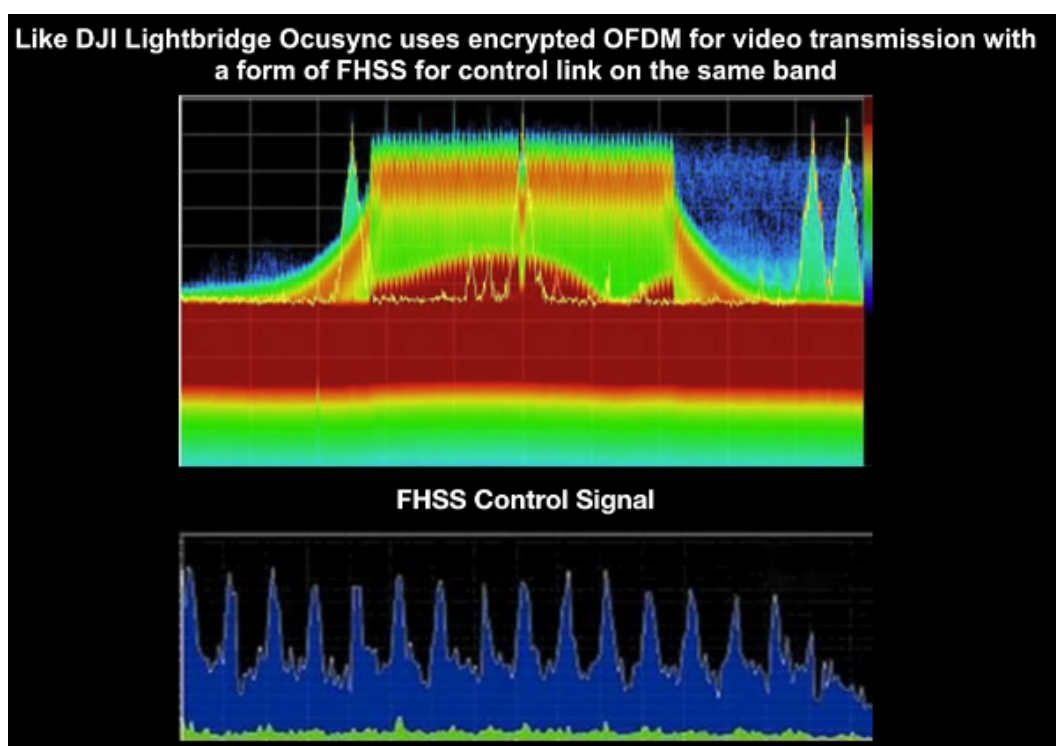
II.3.4. FHSS i OFDM

Zarówno Lightbridge, jak i OcuSync używają szyfrowanej modulacji OFDM (ang. Orthogonal Frequency-Division Multiplexing, zwielokrotnianie z ortogonalnym podziałem częstotliwości)) dla transmisji obrazu i z formy FHSS (ang. Frequency-Hopping Spread Spectrum) dla transmisji sygnałów sterowania. Kanał dla transmisji obrazu nie zmienia się w trakcie całego lotu, pod warunkiem, że nie następują zakłócenia, albo użytkownik nie ustawi ręcznie innej częstotliwości. Z kolei metoda FHSS skacze"po częstotliwościach w całym dostępnym widmie, w tym nawet w pasmie przeznaczonym do transmisji obrazu.[23] [19]



Rys. 14. DJI RC plus

Źródło: <https://store.dji.com>

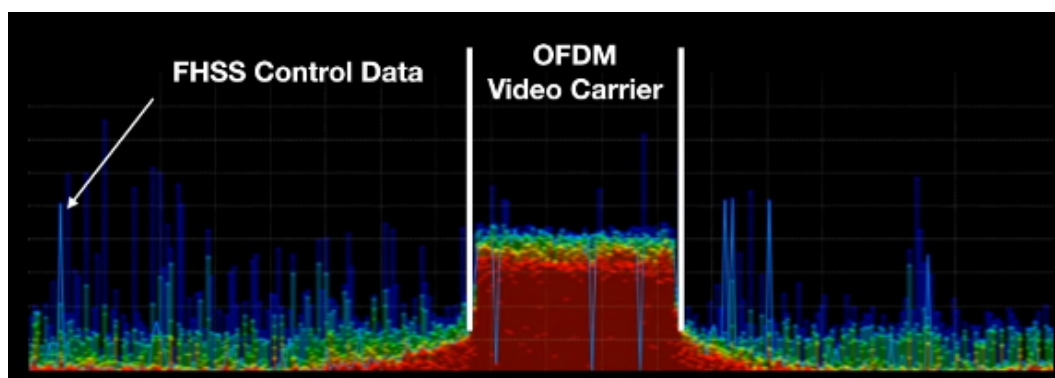


Rys. 15. Widmo OFDM i FHSS

Źródło: <https://www.youtube.com/watch?v=gfqcSv9sR0A>

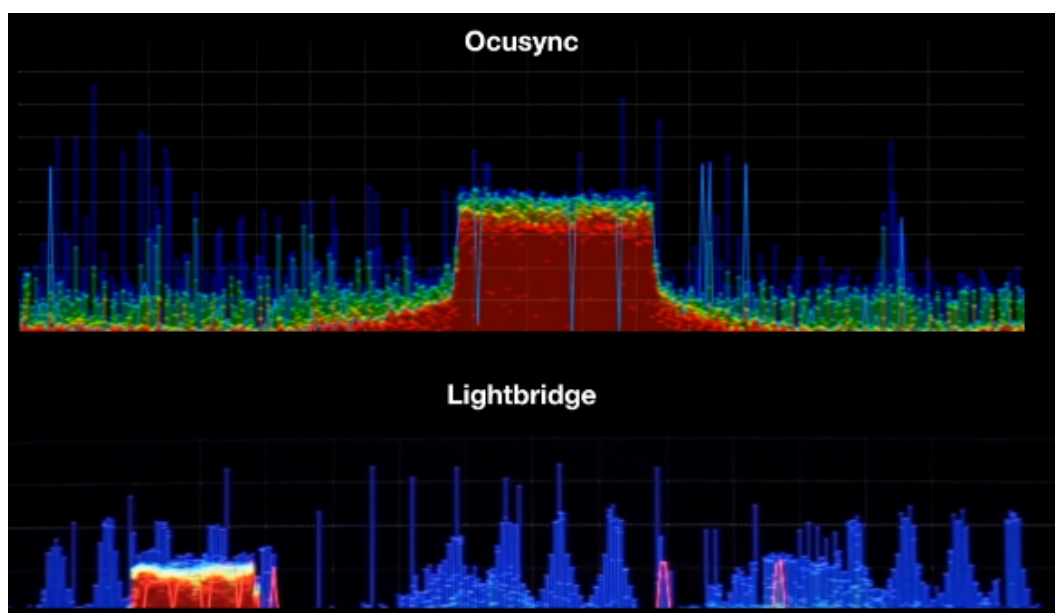
II.3.5. Przewaga OcuSync nad Lightbridge

OcuSync stało się główną technologią rozwijaną przez DJI. Producent wykorzystuje układy scalone przeznaczone do komunikacji WiFi. Wytwarza na nie swoje oprogramowania, która można bez problemu aktualizować. Lightbridge nie miał takiej możliwości, a w dodatku komunikacji opierała się wyłącznie na paśmie 2,4GHz. Nowe procesory



Rys. 16. Widmo OcuSync z zaznaczoną modulacją FHSS i OFDM

Źródło: <https://www.youtube.com/watch?v=gfqcSv9sR0A>



Rys. 17. Porównanie widma OcuSync i Lightbridge

Źródło: <https://www.youtube.com/watch?v=gfqcSv9sR0AV>

w układach WiFi dzięki coraz większej częstotliwości pracy zapewniły osiągnięcie tych samych efektów, co DJI uzyskiwał za pomocą autorskich układów scalonych, bez dodatkowego kosztu wynikającego z produkcji.

II.4. Kontrolowanie BSP za pomocą API dostarczanego od producenta

Przeszukując internet w poszukiwaniu bezzałogowych statków powietrznych umożliwiających ich sterowanie za pomocą API od producenta, można napotkać głównie rozwiązania od DJI. Wszystkie pozostałe rozwiązania nie działają na gotowych dronach, a na oprogramowaniu przeznaczonym do wgrania na wybranych jednostkach do sterowania modelami RC.

Najpopularniejszym tego rozwiązaniem jest ArduPilot, czyli pakiet oprogramowania nawigacyjnego działającego w pojeździe wraz z oprogramowaniem sterującym stacją naziemną.

II.4.1. DJI SDK

DJI dostarcza do swoich produktów następujące interfejsy API:

1. **App Dev.** - interfejsy API przeznaczone do sterowania dronem z poziomu stacji bazowej, kontroler stanowi interfejs pośredniczący między aplikacją wykorzystującą SDK a dronem powietrznym:
 - a) **Mobile SDK** - SDK przeznaczona na platformę iOS i Android. Aplikacja na smartfon za pomocą kabla USB podłączonego do kontrolera statku powietrznego realizuje zaprogramowaną logikę działania.
 - b) **UX SDK** - to Mobile SDK rozszerzony o elementy interfejsu użytkownika, co przyspiesza znacznie proces tworzenia oprogramowania.
 - c) **Windows SDK** - SDK umożliwiające wydawanie aplikacji na systemach operacyjnych Windows.
2. **Payload Dev.** - interfejsy API przeznaczone do nadawania logiki działania drona na poziomie samego drona, dzięki temu po utracie zasięgu może ona dalej funkcjonować. Opcja dostępna dla najdroższych wersji dronów DJI, które można dostosowywać do swoich wymagań za pomocą odpowiednich rozszerzeń, np. kamery termowizyjnej
 - a) **Payload SDK** - zestaw narzędzi programistycznych umożliwiających tworzenie oprogramowania do rozszerzeń, które mogą być montowane na dronach DJI.
 - b) **Onboard SDK** - otwarto-źródłowe API umożliwiające bezpośrednią komunikację z wybranymi dronami i kontrolerami za pomocą interfejsu szeregowego.

Rozdział III. Projekt mobilnego systemu zarządzania i sterowania BSP

III.1. Wymagania funkcjonalne

TODO

III.2. Wymagania pozafunkcjonalne

Tab. 3. Wymagania pozafunkcjonalne

Lp.	Opis
1	Aplikacja musi po zerwaniu połączenia z brokerem MQTT automatycznie ją przywrócić wraz z subskrybowanymi Topic-a.
2	Aplikacja powinna mieć możliwość ustawienia czas po jakim operacje na protokole MQTT zostaną uznane za zakończone niepowodzeniem.
3	Aplikacja musi komunikować się na indywidualnych Topic-ach danego statku powietrznego.
4	Aplikacja powinna w ścieżkach określających Topic-i MQTT zawierać numer seryjny urządzenia.
5	Aplikacja powinna przy komunikacji z brokerem MQTT jako swój identyfikator klienta używać numeru seryjnego urządzenia.
6	Komunikacja MQTT musi być zabezpieczona loginem i hasłem.
7	Aplikacja musi obsługiwać przypadki, gdy otrzymana komenda do wykonania przez BSP nie będzie poprawna.
8	Aplikacja musi automatycznie potwierdzać lądowania urządzenie, jeżeli jest to wymagane.
9	Aplikacja powinna mieć możliwość ustawiania interwału czasowego w jakim zostaną przesłane dane dotyczące statusu statku.
10	Aplikacja powinna przysyłać obrazy w najwyższej dostępnej rozdzielczości.
10	System musi być odporny na przerwy w dostępie do sieci.
11	Aplikacja powinna przysyłać informacje zwrotną dotyczącą wykonania komendy na dedykowany Topic.

Źródło: Badania własne.

III.3. Stos technologiczny

Na wstępie warto zaznaczyć, że większość technologii w stosie technologicznym jest wyłącznie najrozsądniejszą odpowiedzią na wcześniej podjęte decyzje.

Pierwszą decyzją w tym ciągu przyczynowo-skutkowym było wybranie *DJI Mobile SDK*, ponieważ taki wybór pozwoli obsłużyć większość dronów dostępnych na rynku. Biblioteka ta dostarcza szereg możliwości i została ona dobrze udokumentowana. Taki wybór ogranicza nas do dwóch platform mobilnych Android i iOS. Ciężko dostrzec znaczącą przewagę technologiczną w którejkolwiek z nich. Wybór jednak padł na platformę Android, ze względu na doświadczenie autora z nią i praktycznie tańsze urządzenia mobilne.

Ostatnio dużą popularnością w kontekście pisania aplikacji mobilnych zyskuje *Flutter*. Głównie ze względu na swoją wieloplatformowość, dzięki której nie ma potrzeby utrzymywania dwóch kodów przeznaczonych na platformy Android i iOS, a wyłącznie jednej, która działa na obu. Jednak biblioteka dostarczona od DJI jest biblioteką w języku Java, wymusza to na nas pisanie natywnej aplikacji w Androidzie. Tak więc wykorzystanie wcześniej wspomnianego narzędzia nie przyniesie dodatkowych korzyści.

Natywne aplikacje w Androidzie działają na wirtualnej maszynie Javy. Takie ograniczenie pozwala na wykorzystanie również języka programowania Kotlin. Jest on wykonywany w języku Java i zapewnia pełną z nią interoperacyjność. Korzyścią odniesioną z wykorzystania tego języka jest większa kontrola nad wartościami *null*, która wymusza Kotlin. Dostarcza on także szereg narzędzi skracających zapisy kodu, który realizowałby tę samą funkcjonalność w czystej Javie.

Kolejny wybór, jaki należało dokonać to narzędzia do automatycznej budowy oprogramowania. W kontekście Javy wybór ogranicza się praktycznie do *Maven* i *Gradle*. Zarządzania zależnościami i ich obsługa jest identyczna. Wybór padł na *Gradle*, głównie ze względu na osobiste doświadczenie autora i brak sympatii do formatu *xml*, który w *Maven* jest obecny.

Ostatni etap to wybór technologii, za pomocą której będzie odbywała się komunikacja serwera z pojedynczymi BSP. Najpopularniejsze podejście polegające na stworzeniu RestAPI nie jest możliwe, ponieważ nie obsługuje ono przypadku, w którym połączenie zostanie przerwane, a w tego typu projekcie należy zakładać, że takie zakłócenia mogą występować. Tak, więc wybór padł na protokół komunikacyjny MQTT, który dostarcza narzędzia odporne na zakłócenia w połączeniu sieciowym.

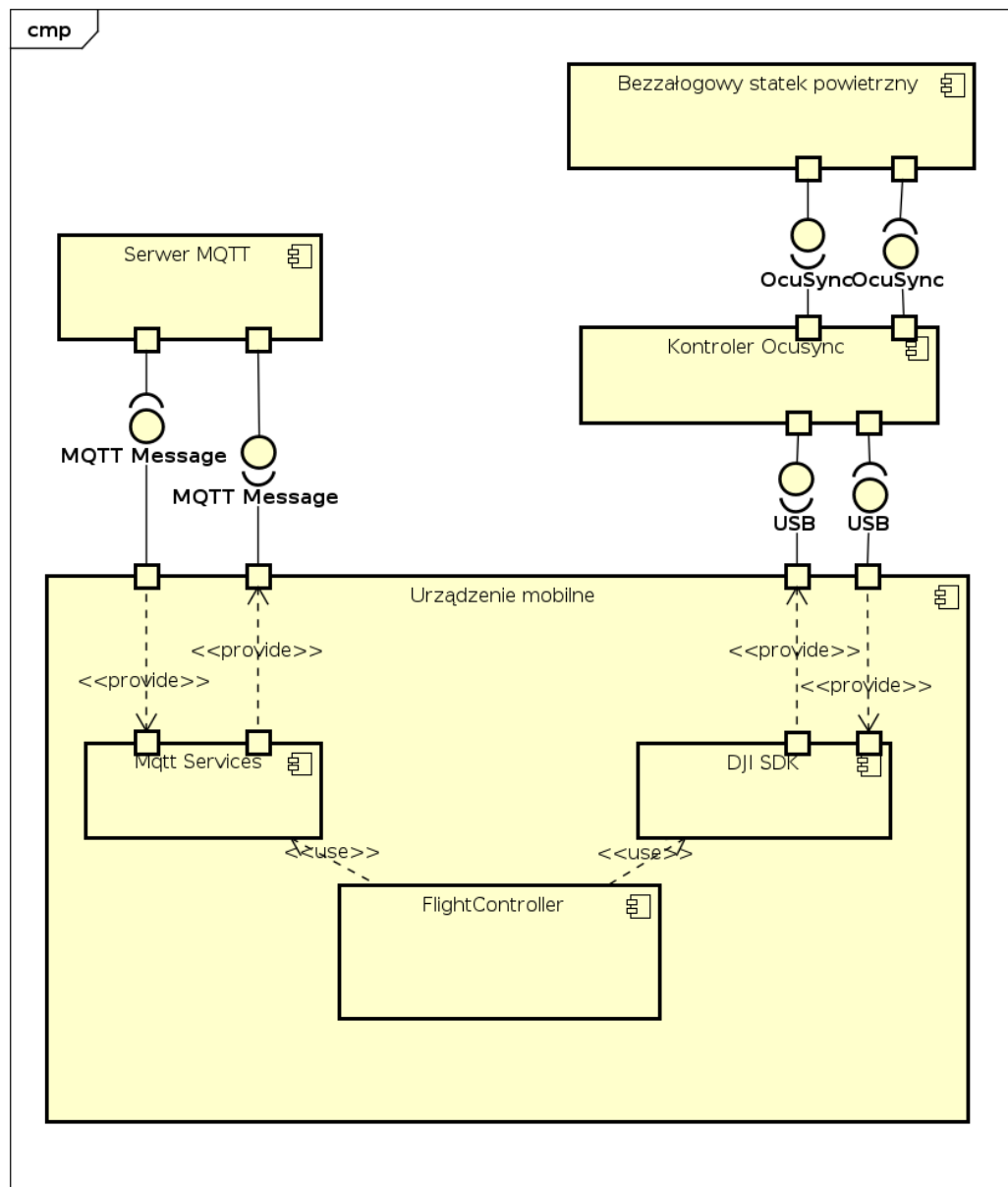
Podsumowując, stos technologiczny prezentuje się następująco:

- DJI Mobile SDK
- Android
- Java
- Gradle
- MQTT

III.4. Wysokopoziomowy diagram systemu

TODO tutaj diagram całości jako rój

III.5. Diagram komponentów



Rys. 18. Diagram komponentów

Źródło: Własne

Komunikacja dronów w ramach systemu odbywa się za pomocą serwera MQTT. Pobiera on dane ze statku i nim steruje za pomocą określonych komend. W ramach urządzenia mobilnego są wyróżnione trzy komponenty:

- **Mqtt Services** to serwisy, które realizują komunikację z brokerem MQTT. Ich projekt i implementacja doży do tego, aby API wystawione do komponentu *FlightController* wymagało jak najmniej działania.

- **DJI SDK** to dwie biblioteki dostarczane od producenta: *DJI SDK* i *DJI UXSDK*. Pierwsza z nich udostępnia metody umożliwiające sterowanie dronem i odczytywanie jego parametrów, a druga dostarcza komponenty w Androidzie umożliwiające obsługę drona za pomocą elementów graficznych.
- **FlightController** korzysta z dwóch wcześniej wymienionych komponentów, w nim zawarta logika kontrolowania drona.

Kontroler Ocusync, a dalej dron, są sterowane za pomocą wspomnianej biblioteki dostarczonej przez producenta.

III.6. Diagramy klas

Poniżej przedstawiono diagramy klasy przedstawiające architekturę systemu.

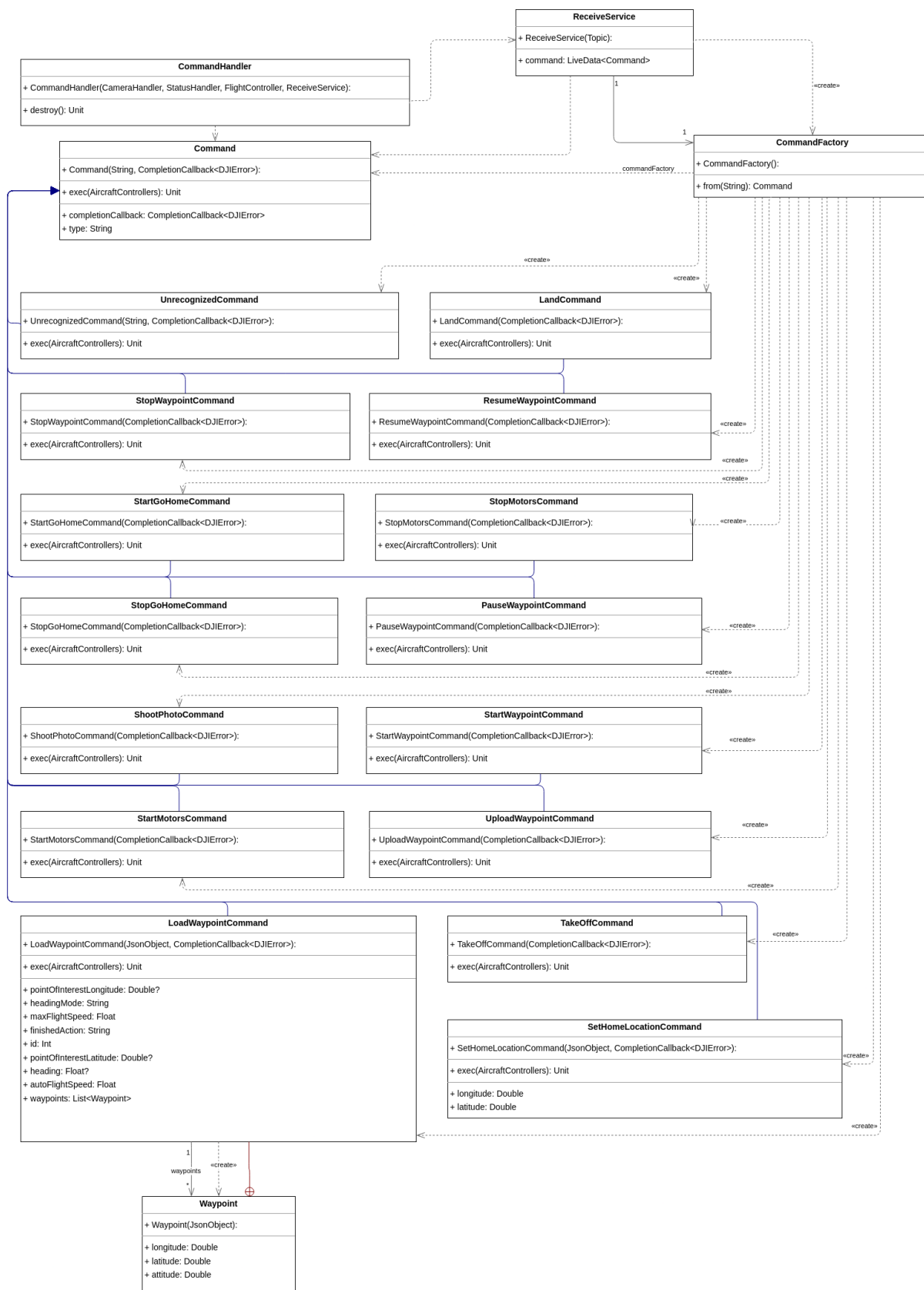
III.6.1. Diagram klas ograniczony do dostępnych komend w ramach systemu

Przy projektowaniu abstrakcji komend, które można wydawać systemowi BSP, można wyróżnić dwa podejścia. Pierwsze z nich, które jest zaprezentowane poniżej (Rys. 19.), polega na przedstawianiu każdej z nich w postaci osobnej klasy z przyrostkiem *Command*. Dziedziczą one po klasie *Command*, która posiada abstrakcyjną metodę *exec*. Celem tej metody jest zrealizowanie polecenia tożsamego z daną komendą, za pomocą obiektów dostarczonych przez DTO *AircraftControllers*.

Na diagramie przedstawiono również dwie istotne klasy *CommandFactory* i *CommandHandler*. Celem pierwszej z nich jest zwrócenie odpowiedniej instancji klasy dziedziczącej po *Command*, na podstawie wartości tekstowej w formacie JSON. *CommandHandler* odpowiada za odbieranie tych wartości za pomocą serwisu *ReceiveService*, a następnie zmapowanie jej za pomocą *CommandFactory* i wywołanie metody *exec* z klasy *Command*.

Wspomniane wcześniej drugie podejście polegałoby na przedstawieniu tych komend w postaci wartości typu wyliczeniowego (enum). Jednak należy wtedy przyjrzeć się, jak wyglądałoby dodawanie kolejnych komend do systemu. Obecnie, aby dodać nową komendę do systemu należy otworzyć nową klasę, która dziedziczy po *Command*, a następnie dodanie jej przypadku do *CommandFactory*. Modyfikujemy kod w ten sposób w tylko jednym miejscu. Klasa *CommandHandler*, która posiada dużą odpowiedzialność, pozostanie niezmieniona. W przypadku, gdyby zastosowano podejście z typem wyliczeniowym, dodanie kolejnej komendy do systemu, wiązałoby się z modyfikacją, oprócz wcześniej wspomnianej *CommandFactory*, klasy *CommandHandler*, która wymagałaby dodania kolejnej instrukcji wyboru. Dodatkowo w klasie znajdowałoby się 15 metod, które nie są do końca związane z jej odpowiedzialnością. Nawet jeżeli wynieść takie metody do osobnej klasy np. *CommandExecutor*, jej odpowiedzialność nadal byłaby bardzo duża.

Roberta C. Martina, autor książki *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall., która często bywa określana jako "biblia programistów", przedstawił zasady dobrego programowania obiektowego, określane jako SOLID. Podejście, w którym architektura komend opierałaby się na typie wyliczeniowym, stanowiłoby naruszenie przynajmniej dwóch zasad *Single responsibility principle* i *Open/closed principle*. Przytoczony przykład ukazuje, że przedstawione rozwiązanie jest, jak najbardziej poprawne.[38][39]



Rys. 19. Diagram klas ograniczony do dostępnych komend w ramach systemu

Źródło: Własne

III.6.2. Diagram klasy służącej do przechowywania stanu BSP

Diagram klas(Rys. 20.) przedstawia *FlightStatus*, która przechowuje dane dotyczące statusu BSP w danej chwili czasu. Posiada ona tylko metodę *toJson*. Zwraca ona stan obiektu w formacie JSON, które przesyłany jest za pomocą serwisu MQTT. Klasy, które nie realizują żadnej logiki, a służą do przechowywania danych, określa się jako DTO (Data Transfer Object).



Rys. 20. Diagram klasy służącej do przechowywania stanu BSP

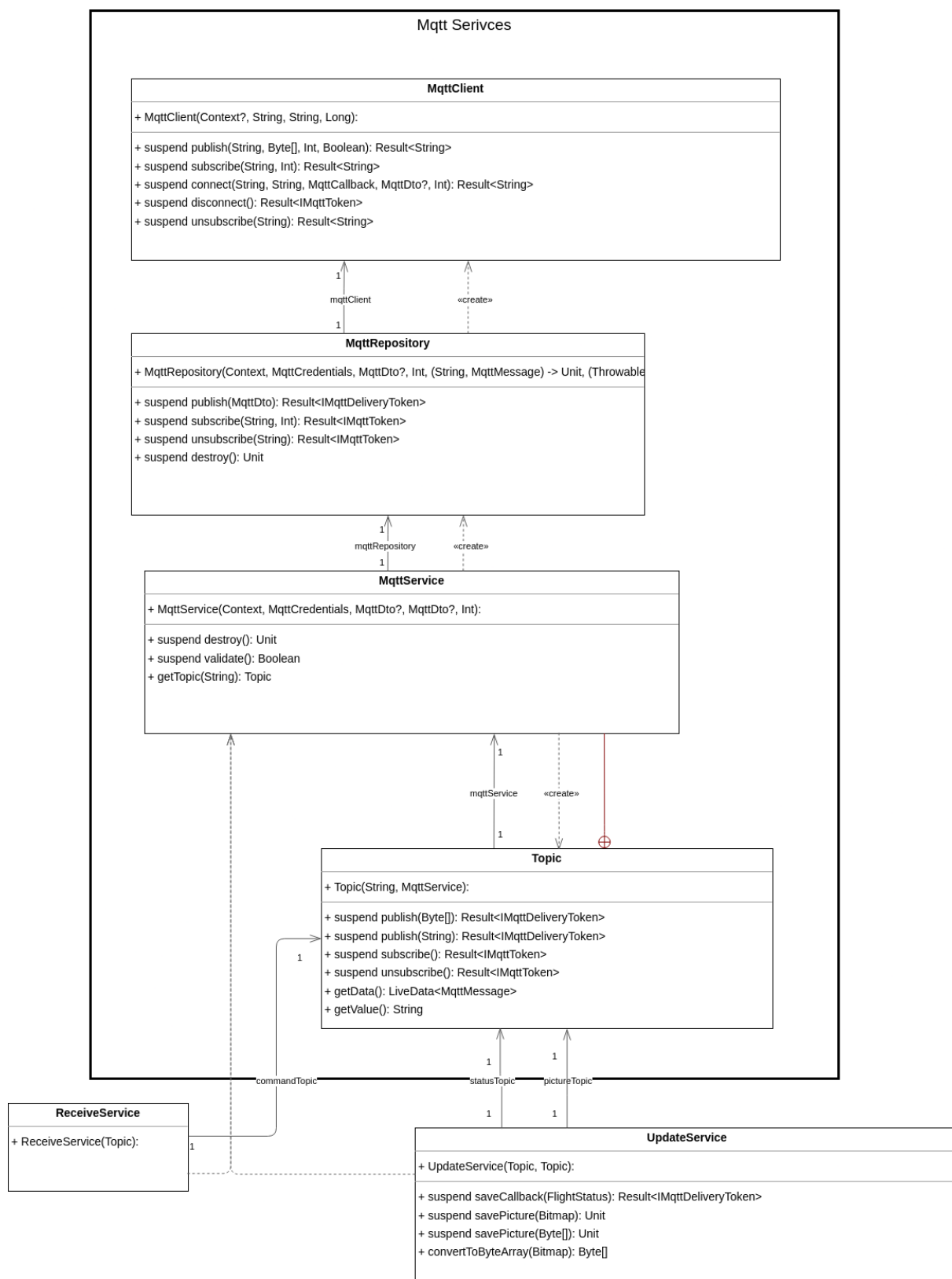
Źródło: Własne

III.6.3. Diagram klas ograniczony do serwisów działających w ramach komunikacji MQTT

Poniższy diagram klas (Rys. 21.) reprezentuje *Mqtt Services* z wcześniej przedstawionego diagramu komponentów (Rys. 18.). Z punktu widzenia *FlightController* dostępne są wyłącznie metody z klasy *MqttService*. Najważniejsza z nich to *getTopic*. Dostarcza ona abstrakt Topic-u MQTT, tj. instancje klasy *Topic*. Dostarcza ona szereg metod, które umożliwiają komunikację na wybranym Topic-u. Metoda *getData* zwraca obiekt typu *LiveData*, który służy w systemie android do przechowywania zmieniających się danych, za pomocą klasy *Observer* dodaje się wyrażenie lambda, które zostają wywołane za każdym razem, gdy obiekt zostanie zmodyfikowany. Obiekt otrzymuje dane z brokera tylko i wyłącznie jeżeli wcześniej włączy się subskrybuje danego Topica metoda *subscribe*.

W ramach systemu zostaną wykorzystane następujące Topic-ki MQTT:

1. Wykorzystywane wewnątrz komponentu *Mqtt Services*:
 - a) Na Topic ***droman/birth*** przy połączeniu z brokerem MQTT zostanie zapisany identyfikator klienta;
 - b) Przy zerwaniu połączenia na ***droman/lastWill*** zostanie zapisany identyfikator klienta;
 - c) Do walidacji połączenia posłuży Topic ***droman/validate***. Jeżeli *MqttService* nie jest w stanie opublikować na podanym Topic-u wiadomości, połączenie zostanie uznane za niepoprawne;
2. Wykorzystywane przez serwisy *ReceiveService* i *UpdateService*
 - a) Na Topic ***droman/status/{clientId}*** z pewnym interwałem czasowym są zapisywane dane określające stan BSP w formacie JSON;
 - b) Urządzenie mobilne nasłuchuje na Topic-u ***droman/command/{clientId}***, na którego wysyłane są polecenia do zrealizowania przez pojedynczy BSP.
 - c) Rejestrowane obrazy są przesyłane na ***droman/picture/{clientId}*** w formacie *jpeg*.

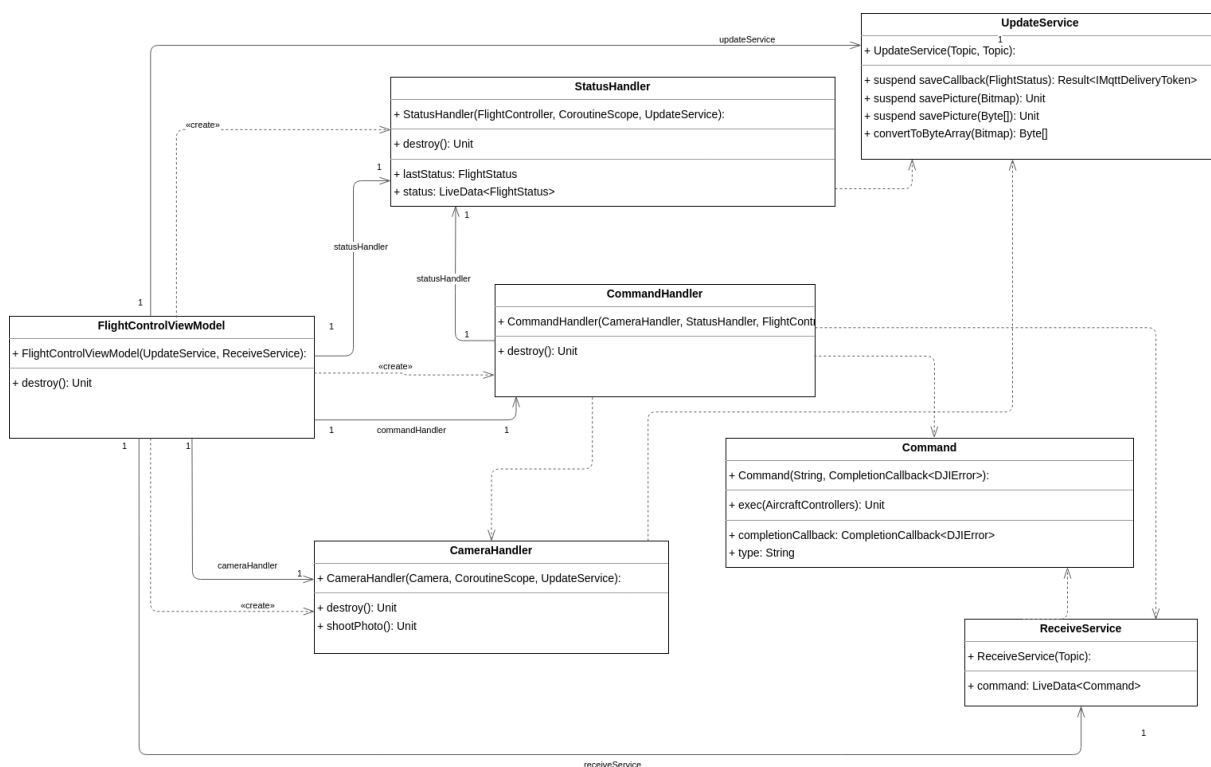


Rys. 21. Diagram klas ograniczony do serwisów działających w ramach komunikacji MQTT

Źródło: Własne

III.6.4. Diagram klas ograniczony do obszaru kontroli lotu

Przedstawione klasy w poniższym diagramie (Rys. 22.) stanowią *FlightController* z wcześniejszego diagramu komponentów (Rys. 18.). Wyłącznie klasy z przyrostkiem *Handler* wywołują metody na udostępnionej bibliotece do obsługi drona on DJI. Podział na takie trzy klasy ma na celu zmniejszenie odpowiedzialności, zgodnie z zasadą SOLID *Single responsibility principle*. *FlightControlViewModel* jest ściśle powiązany z *Activity* dotyczące sterowania z Andoridem. Dwa pozostałe serwisy *UpdateService* i *ReceiveService* służą do kolejno zapisywania i odczytywania wybranych Topic-ów.



Rys. 22. Diagram klas ograniczony do obszaru kontroli lotu

Źródło: Własne

III.7. Wykorzystane urządzenia



Rys. 23. DJI Mini 2

Źródło: <https://laptrinhx.com/dji-mini-2-review-same-compact-size-more-confidence-flying-3163139258/>

W trakcie implementacji i testów zostaną wykorzystane następujące urządzenia:

- **Laptop**, który posłuży jako serwer MQTT, a także do wykonywania skryptów wysyłających do systemu określone komendy.
- **Dron DJI Mini 2 wraz z kontrolerem**, czyli produkt docelowy, którego sterowaniem ma umożliwiać zaimplementowany system.
- **Smartfon z systemem Android**, który będzie podłączony do kontrolera DJI. Aplikacja mobilna na tym urządzeniu będzie odbierała komendy z serwera MQTT, które będą definiowały jak sterować dronem. Ponadto będzie na bieżąco przysyłała status drona do serwera. Wszystko to będzie odbywało się na unikalnym topicu MQTT, który będzie zawierał numer seryjny urządzenia.



Rys. 24. Kontroler do DJI Mini 2

Źródło: <https://www.gohero.pl/userdata/public/gfx/6191/Kontroler-do-DJI-Mavic-Air-2.jpg>

Rozdział IV. Implementacja systemu

W poniższym rozdziale zaprezentowano, wyjaśniono i uzasadniono działanie najważniejszych fragmentów kodu systemu. W ostatnim podrozdziale zaprezentowano interfejs użytkownika na urządzeniu mobilnym.

IV.1. Wykonywanie komend

Poniżej zaprezentowano implementacje klas, które służą do wykonywania poleceń w systemie. `textitCommand` (Kod. 1.) jest klasą, po której dziedziczą wszystkie komendy, które mogą być wykonane w systemie. Jej kluczowym elementem jest metoda `exec`, która w klasach potomnych służy do zrealizowania polecenia tożsamego z jej nazwą. Jedyne argumenty w tej metodzie, jest typu `AircraftControllers`. Jest to DTO dla wszystkich obiektów, które są wymagane do zrealizowania komendy.

Jednym z przykładów jest klasa `StartMotorsCommand` (Kod. 2.), która odpowiada zgodnie z jej nazwą, za wystartowanie silników. W linii nr 7 za pomocą `AircraftControllers` wyciągany jest ostatni status BSP. Następnie w linii nr 6 dokonywana jest sprawdzenie, czy BSP nie lata i nie ma włączonych silników, jeżeli nie, to za pomocą klasy `FlightController` z biblioteki DJI uruchamiane są silniki. W przeciwnym wypadku zwracany jest stosowny komunikat (linia nr 13).

Kod. 1. Klasa *Command*

```

1  abstract class Command(val type: String, val completionCallback: CommonCallbacks.
    CompletionCallback<DJIError>) {
2      abstract fun exec(aircraftControllers: AircraftControllers)
3
4      companion object {
5          const val TAG = "Command"
6      }
7  }
```

Źródło: Badania własne.

Kod. 2. Klasa *StartMotorsCommand*

```

1  class StartMotorsCommand(completionCallback : CommonCallbacks.CompletionCallback<
    DJIError>) : Command(type, completionCallback) {
2      companion object {
3          const val type = "start_motors"
4      }
5
6      override fun exec(aircraftControllers: AircraftControllers) {
7          val status = aircraftControllers.statsHandler.getLastStatus()
8          if (!status.isFlying && !status.motorsOn) {
9              aircraftControllers.flightController.turnOnMotors(
10                 completionCallback
11             )
12          } else {
13              FeedbackUtils.setResult("Forbidden_state_can't_start_motors", TAG)
14          }
15      }
16  }
```

Źródło: Badania własne.

Samo wykonywanie metody `exec` ogranicza się w klasie `CommandHandler` do 3 linii kodu (Kod. 3.), jest to korzyść odniesiona dzięki dobrze dobranej architekturze.

Pozostałe ciało klasy zostanie omówione na późniejszych stronach.

Kod. 3. Fragment kodu z CommandHandler

```

1 private val commandObserver = Observer<Command> {
2     it.exec(aircraftControllers)
3 }

```

Źródło: Badania własne.

CommandFactory (Kod. 4.) zgodnie ze wzorcem projektowym *Factory* (pl. *Fabryka*), odpowiada za tworzenie kolejnych instancji klasy *Command*. Metoda *getCompletionCallback* dostarcza uniwersalny obiekt do obsługi poleceń wykonywanych za pomocą metod z biblioteki DJI. Najważniejsza jest jednak w niej metoda *from*. Dla przedstawionej wartości tekstowej w formacie JSON zwraca ona instancję odpowiedniej klasy. W linii nr 10 za pomocą biblioteki *Gson* tworzony jest obiekt *JsonObject*, na podstawie zawartości zmiennej *value*. Jeżeli wartość tekstowa nie jest w ogóle w formacie JSON, zostanie rzucony wyjątek, którego obsługa polega na zwróceniu instancji *UnrecognizedCommand* (linia nr 15), zgonie z wcześniej przyjętą zasadą, która mówi, że zwrócenie wartości *null* jest złą praktyką. Metoda *createCommand* na podstawie dostarczonych argumentów zwraca odpowiednią instancję klasy *Command*, to w jej ciele zostanie dokonana zmiana, jeżeli do systemu zostanie dodana kolejna komenda. Klasa ta może rzucać wyjątki (linia nr 23). Jest to spowodowane tym, że w momencie tworzenia niektórych komend, które zawierają dodatkowe parametry, jak np. współrzędne dokonywana jest walidacja ich poprawności. Jeżeli te wartości będą niepoprawne, zostanie rzucony wyjątek, ponieważ nie można dopuścić do wykonania polecenia z niepoprawnymi parametrami. Również on jest obsługiwany w liniach nr 13-16.

Kod. 4. Klasa *CommandFactory*

```

1 class CommandFactory {
2     companion object {
3         const val TAG = "CommandFactory"
4     }
5
6     private val gson: Gson = Gson()
7
8     fun from(value: String): Command {
9         return try {
10             val jsonObject = gson.fromJson(value, JsonObject::class.java)
11             val missionType = jsonObject.get("type").asString
12             createCommand(missionType, jsonObject)
13         } catch (e: Exception) {
14             FeedbackUtils.setResult(e.toString(), level = LogLevel.ERROR, tag = TAG)
15             UnrecognizedCommand(value, getCompletionCallback(UnrecognizedCommand.type))
16         }
17     }
18
19     private fun getCompletionCallback(type: String): CompletionCallbackImpl<DJIError> {
20         //returns completion callback object
21     }
22
23     @Throws(Throwable::class)
24     private fun createCommand(type: String, jsonObject: JsonObject): Command {
25         return when (type) {
26             StartGoHomeCommand.type -> {
27                 StartGoHomeCommand(getCompletionCallback(StartGoHomeCommand.type))
28             }
29             LandCommand.type -> {
30                 LandCommand(getCompletionCallback(LandCommand.type))
31             }
32             ShootPhotoCommand.type -> {

```

```

33         ShootPhotoCommand(getCompletionCallback(ShootPhotoCommand.type))
34     }
35     // ...
36     //and more cases
37     // ..
38     PauseWaypointCommand.type -> {
39         PauseWaypointCommand(getCompletionCallback(PauseWaypointCommand.type))
40     }
41     else -> {
42         UnrecognizedCommand(
43             jsonObject.toString(),
44             getCompletionCallback(UnrecognizedCommand.type)
45         )
46     }
47 }
48 }
49
50 }

```

Źródło: Badania własne.

IV.2. Obsługa MQTT

W tej sekcji zaprezentowano klasy związane z obsługą komunikacji za pomocą MQTT. Na poziomie najbliższym komponentowi *FlightController* (Rys. 18.) ważne są wyróżnienia klasy *MqttService* (Kod. 6.) i jej klasa wewnętrzna *Topic* (Kod 5.).

Ta wewnętrzna klasa dostarcza metody, które umożliwiają komunikację na wybranych Topic-ach z minimalnym wysiłkiem dla programisty. Jako argument w swoim konstruktorze przyjmuje ścieżkę Topica MQTT. Dzięki temu metody odpowiedzialne za publikowanie wiadomości MQTT (linie nr 2-8) ograniczone są do jednego argumentu, ich treści. Pozostałe metody, jak np. włączenia subskrypcji (linie 14-15), sprawdzenia czy dany Topic jest subskrybowany (10-12) czy pobierania wiadomości nie potrzebują żadnych dodatkowych argumentów. Klasa ta też jest bardzo krótka ponieważ wszystkie skaplikowane operacje są wykonywane przez *MqttService*.

W której to (Kod. 6.) należy wyróżnić dwie istotne zmienne *subscribedTopics* i *messagesArrived* (linie 9-10). Pierwsza z nich to zbiór wartości tekstowych, konkretnie ścieżek, które są subskrybowane przez klienta Mqtt. Kolejne wartości są dodawane do niego wraz z rozpoczęciem subskrypcji danego Topica w metodzie *subscribe* (linie 48-54), a usuwane w momencie zakończenia subskrypcji metodą *unsubscribe* (linie 40-41).

Przy opisywaniu zmiennej *messagesArrived* należy zwrócić uwagę na wyrażenie lambda *messageArrivedFun* (linie 16-23), które jest wykonywane za każdym razem gdy klient MQTT otrzyma wiadomość. W jego ciele najistotniejsza jest linia nr 20, w trakcie której to za pomocą metody *getTopicData* pobierany jest obiekt *LiveData<MqttMessage>* dotyczący danego Topica, a następnie zapisywana jest nowa wiadomość za pomocą metody *postValue*. Analizując jeszcze metodę *getTopicData* należy zwrócić uwagę na linie nr 37, w której to z wcześniej wspomnianej mapy *messagesArrived* pobierana jest metodą *getOrPut* wartość dla klucza równego ścieżce Topic-a, jeżeli w mapie nie ma wartości dla podanego klucza to do mapy dodawana jest nowa para o wartości *MutableLiveData()*. W ten sposób w *messagesArrived* znajdują się pary w których klucz to ścieżka danego Topic-a, który istnieje w systemie, a wartość to obiekt *LiveData<MqttMessage>* z ostatnią otrzymaną wiadomością.

Kod. 5. Klasa wewnętrzna *Topic*

```

1  class Topic(private val value: String, private val mqttService: MqttService) {
2      suspend fun publish(payload: ByteArray): Result<IMqttDeliveryToken> {
3          return mqttService.publish(MqttDto(value, payload))
4      }
5
6      suspend fun publish(payload: String): Result<IMqttDeliveryToken> {
7          return mqttService.publish(MqttDto(value, payload))
8      }
9
10     fun isSubscribed(): Boolean {
11         return mqttService.getSubscribed().contains(value)
12     }
13
14     suspend fun subscribe(): Result<IMqttToken> {
15         return mqttService.subscribe(value)
16     }
17
18     suspend fun unsubscribe(): Result<IMqttToken> {
19         return mqttService.unsubscribe(value)
20     }
21
22     fun getData(): LiveData<MqttMessage> {
23         if (!isSubscribed()) {
24             Log.w(TAG, "getData_without_subscribing_topic_$value")
25         }
26         return mqttService.getTopicData(value)
27     }
28
29     fun getValue(): String {
30         return value
31     }
32 }
33
34

```

Źródło: Badania własne.

Kod. 6. Klasa *MqttService*

```

1  class MqttService(
2      context: Context,
3      mqttCredentials: MqttCredentials,
4      lastWill: MqttDto? = null,
5      birth: MqttDto? = null,
6      keepAliveInterval: Int = GlobalConfig.KEEP_ALIVE_INTERVAL,
7  ) {
8
9      private val subscribedTopics: MutableSet<String> = HashSet()
10     private val messagesArrived: HashMap<String, MutableLiveData<MqttMessage>> =
11         HashMap()
12
13     private val connectionLostFun: (throwable: Throwable) -> Unit = {
14         //... reconnect and resubscribe topics
15     }
16     private val deliveryCompleteFun: (token: IMqttDeliveryToken) -> Unit = { //... }
17
18     private val messageArrivedFun: (topic: String, message: MqttMessage) -> Unit = {
19         topic, message ->
20         if (subscribedTopics.contains(topic)) {
21             Log.d(TAG, "Receive_message_from_topic_$topic")
22             getTopicData(topic).postValue(message)
23         } else { //... some log info }
24     }
25
26     private val mqttRepository = MqttRepository(//... init with values above)
27

```

```

27 init {
28     this.let { service ->
29         DjiApplication.mainScope.launch(Dispatchers.IO) {
30             birth?.let { service.publish(it) }
31         }
32     }
33 }
34
35 private fun getTopicData(topic: String): MutableLiveData<MqttMessage> {
36     return messagesArrived
37         .getOrPut(topic) { MutableLiveData() }
38 }
39
40 private suspend fun unsubscribe(value: String): Result<IMqttToken> {
41     val res = mqttRepository.unsubscribe(value)
42     if (res.isSuccess) {
43         subscribedTopics.remove(value)
44     }
45     return res
46 }
47
48 private suspend fun subscribe(value: String): Result<IMqttToken> {
49     val res = mqttRepository.subscribe(value)
50     if (res.isSuccess) {
51         subscribedTopics.add(value)
52     }
53     return res
54 }
55
56 private fun getSubscribed(): Set<String> {
57     return subscribedTopics
58 }
59
60 private suspend fun publish(data: MqttDto): Result<IMqttDeliveryToken> {
61     return mqttRepository.publish(data)
62 }
63
64 fun getTopic(value: String): Topic {
65     return Topic(value = value, this)
66 }
67
68 suspend fun destroy() {
69     mqttRepository.destroy()
70 }
71
72 suspend fun validate(): Boolean {
73     return mqttRepository.publish(MqttDto(ETopic.VALIDATE, "validate" + UUID.
74         randomUUID()))
75         .isSuccess
76 }
77 class Topic(private val value: String, private val mqttService: MqttService) {
78     //... inner class
79 }
80 }

```

Źródło: Badania własne.

IV.3. Kontrolowanie BSP

Robert C. Martin w swojej książce *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. mówi między innymi o tym że ciało klasy powinno być jak najkrótsze, nie tylko pod względem liczby linii kodu, ale również odpowiedzialności

[39], dlatego też odpowiedzialności w ramach kontrolowania BSP podzielono na 3 klasy *CameraHandler*, *StatusHandler* i *CommandHandler*.

Cyklem ich życia zarządza *FlightControlViewModel* (Kod. 7.), który jest ściśle związany z aktywnością Andorida *FlightControlActivity*. W jego konstruktorze (linie 14-28) inicjowane są klasy na które została podzielona odpowiedzialność. Nie udało się wynieść tego do *FlightControlViewModelFactory* ponieważ wymagają one w swoim ciele kontekstu Adroida, który nie jest dostępny na poziomie *ViewModelProvider.Factory*.

Kod. 7. Klasa *FlightControlViewModel*

```

1 class FlightControlViewModel(
2     private var updateService: UpdateService,
3     private var receiveService: ReceiveService
4 ) : ViewModel() {
5
6     private var cameraHandler: CameraHandler? = null
7     private var statusHandler: StatusHandler? = null
8     private var commandHandler: CommandHandler? = null
9
10    companion object {
11        const val TAG = "FlightControlViewModel"
12    }
13
14    init {
15        DjiApplication.aircraftInstance?.let { aircraft ->
16            FeedbackUtils.setResult("Aircraft_found", LogLevel.DEBUG, TAG)
17            statusHandler = StatusHandler(aircraft.flightController, viewModelScope,
18                updateService)
19            cameraHandler = CameraHandler(aircraft.camera, viewModelScope, updateService)
20            commandHandler = CommandHandler(
21                cameraHandler!!,
22                statusHandler!!,
23                aircraft.flightController,
24                receiveService
25            )
26        } ?: run {
27            FeedbackUtils.setResult("Aircraft_not_found", LogLevel.WARN, TAG)
28        }
29
30    fun destroy() {
31        commandHandler?.destroy()
32        cameraHandler?.destroy()
33        statusHandler?.destroy()
34    }
35    }

```

Źródło: Badania własne.

Klasa *StatusHandler* odpowiada za zapisywanie statusu BSP co pewien interwał czasowy. Obiekt *flightController* 10 razy na sekundę wywołuje wyrażenie lambda z przekazane przez *setStateCallback* w liniach nr 16-28. Żeby ograniczyć liczbę przesyłanych wiadomości MQTT wprowadzono mechanizm, który przy każdym wywołaniu wyrażenia, zwiększa zmienną *lastUpdateCnt* i gdy osiągnie ona wartość określoną przez *GlobalConfig.STATE_RATE_LIMIT* wysyła, następuje jej wyzerowanie a następnie wysłanie wiadomości MQTT. *setStateCallback* działa na obiekcie klasy *FlightControllerState* z biblioteki DJi, dlatego też dokonujemy przekształcenia na klasę z tego systemu *Flight-Status*, m.in. dlatego że udostępnia ona metodę do generowania statusu jako wartości tekstowej w formacie JSON. Dodatkowo w klasie przechowywany jest ostatni status BSP (linia nr 19).

Kod. 8. Klasa *StatusHandler*

```

1 class StatusHandler(
2     private val flightController: FlightController,
3     viewModelScope: CoroutineScope,
4     updateService: UpdateService
5 ) {
6     companion object {
7         const val TAG = "StatusHandler"
8     }
9
10    private var lastUpdateCnt = 0
11    private lateinit var lastStatus: FlightStatus
12    private var _status: MutableLiveData<FlightStatus> = MutableLiveData()
13    var status: LiveData<FlightStatus> = _status
14
15    init {
16        flightController
17            .setStateCallback { state ->
18                viewModelScope.launch(Dispatchers.IO) {
19                    lastStatus = FlightStatus.gen(state)
20                    if (lastUpdateCnt == 0) {
21                        updateService.saveCallback(lastStatus)
22                    }
23                    _status.postValue(lastStatus)
24                    lastUpdateCnt++
25                    lastUpdateCnt %= GlobalConfig.STATE_RATE_LIMIT
26                }
27            }
28    }
29
30    fun getLastStatus(): FlightStatus {
31        return status.value!!
32    }
33
34    fun destroy() {
35        flightController.setStateCallback(null)
36    }
37
38 }

```

Źródło: Badania własne.

Klasa *CameraHandler* (Kod. 9.) odpowiada za operacje związane z aparatem BSP. Udostępnia ona jedną publiczną metodę *shootPhoto*, której wykonanie wywołuje zdjęcie (linie nr 36-41). W konstruktorze tak jak we wcześniej klasie na każde wygenerowanie obrazu ustawiane jest wyrażenie lambda (linie nr 15-33), które pobiera w najwyższej możliwej rozdzielczości obraz i przesyła go za pomocą *UpdateService*. Ponieważ nie wszystkie drony od DJI wspierają tryb przesyłania obrazu w wysokiej jakości metoda *initCheckingIfSupportMediaDownloadMode* sprawdza, czy ustawienie takiego trybu pracy kamery jest możliwe i ustawia odpowiednią wartość zmiennej *supportDownloadMediaMode*. Na koniec warto zwrócić na *MediaFile.getSusPreview*. Jest to rozszerzenie metody, które zwraca obraz zrobionego zdjęcia w postaci obiektu *Bitmap*. Ponieważ jest to metoda asynchroniczna, nie należy przejmować się jej wstrzymaniem (linia nr 57) w oczekiwaniu na wczytanie obrazu w postaci bitmapy.

Kod. 9. Klasa *CameraHandler*

```

1 class CameraHandler(
2     private val camera: Camera,
3     viewModelScope: CoroutineScope,
4     updateService: UpdateService
5 ) {

```

```

6  companion object {
7      const val TAG = "CameraHandler"
8  }
9
10 private var supportDownloadMediaMode = false
11
12 init {
13     initCheckingIfSupportMediaDownloadMode()
14
15     camera.setMediaFileCallback {
16         if (supportDownloadMediaMode) {
17             viewModelScope.launch(Dispatchers.IO) {
18                 it.getFullView()
19                     ?.let { byte -> updateService.savePicture(byte) }
20             }
21         } else {
22             viewModelScope.launch(Dispatchers.IO) {
23                 it.getSusPreview()?.let { it1 -> updateService.savePicture(it1) }
24                 ?: run {
25                     FeedbackUtils.setResult(
26                         "Cannot_get_preview",
27                         level = LogLevel.ERROR,
28                         tag = TAG
29                     )
30                 }
31             }
32         }
33     }
34 }
35
36 fun shootPhoto() {
37     camera.startShootPhoto(
38         CompletionCallbackImpl<DJIError>(TAG,
39             { FeedbackUtils.setResult("Success_shoot_photo", TAG) })
40     )
41 }
42
43 fun initCheckingIfSupportMediaDownloadMode() { //... }
44
45 private fun setBackToShootPhotoMode() { //... }
46
47 private suspend fun MediaFile.getSusPreview(): Bitmap? {
48     if (this.preview != null) {
49         return this.preview
50     }
51     this.fetchPreview(
52         CompletionCallbackImpl<DJIError>(TAG,
53             { FeedbackUtils.setResult("Success_fetching_photo_preview", TAG) })
54     )
55     var inc = 0
56     while (this.preview == null && inc < 200) {
57         delay(100)
58         inc++
59     }
60     return this.preview
61 }
62
63 private suspend fun MediaFile.getFullView(): Bitmap? {
64     val destDir =
65         File("${Environment.getExternalStorageDirectory().path}/${GlobalConfig.
66             FOLDER_FOR_HQ_MEDIA}/")
67     val downloadHandler = DownloadListenerImpl<String>(camera)
68     this.fetchFileData(destDir, this.fileName, DownloadListenerImpl<String>(camera))
69     return downloadHandler.getBitmap()
70 }
71 fun destroy() {

```

```

72     camera.setMediaFileCallback(null)
73 }
74 }

```

Źródło: Badania własne.

Ostatnia klasa *CommandHandler* (Kod. 10.) odpowiada za realizowanie otrzymywanych komend przez BSP. Ze względu na wyniesie logiki wykonywania poleceń do klasy *Command* klasa ta mocno skróciła swoje ciało i też dzięki temu jest bardziej czytelna. *commandLiveData* jest to obiekt *LiveData<Command>*, który jest wynikiem subskrybowania Topic-u przeznaczonego do przesyłania poleceń i odpowiedniego prze mapowania tych danych przez *ReceiveService*. Bardzo istotna w tej klasie jest zarządzanie cyklem życia obiektów typu *Observer*. Cykl wykonywania wyrażeń zawartych w *commandObserver* i *statusObserver* rozpoczynany jest w konstruktorze tej klasy (linie nr 35-36) i zakańczany w momencie wywoływania metody *destroy* (linie 49-56). Dodatkowo w tych samych miejscach (linie nr 37-40 i 53-55) włączany i wyłączany jest tryb symulacji drona na podstawie *GlobalConfig.SIMULATOR_MODE*.

Kod. 10. Klasa *CommandHandler*

```

1  class CommandHandler(
2      cameraHandler: CameraHandler,
3      private val statsHandler: StatusHandler,
4      private val flightController: FlightController,
5      receiveService: ReceiveService,
6  ) {
7      companion object {
8          const val TAG = "MissionHandler"
9      }
10     private val waypointMissionOperator = MissionControl.getInstance().
        waypointMissionOperator
11     private val waypointMissionOperatorListener = WaypointMissionOperatorListenerImpl()
12
13     private val commandLiveData: LiveData<Command> = receiveService.getCommand()
14
15
16     private val commandObserver = Observer<Command> {
17         it.exec(aircraftControllers)
18     }
19
20     private val aircraftControllers = AircraftControllers(
21         cameraHandler,
22         statsHandler,
23         flightController,
24         waypointMissionOperator
25     )
26     private val statusObserver = Observer<FlightStatus> {
27         if (it.isLandingConfirmationNeeded) {
28             flightController.confirmLanding(
29                 CompletionCallbackImpl<DJIError>( //... feedback flow )
30             )
31         }
32     }
33
34     init {
35         commandLiveData.observeForever(commandObserver)
36         statsHandler.status.observeForever(statusObserver)
37         if (GlobalConfig.SIMULATOR_MODE) {
38             flightController.simulator
39                 .start( // ... initializationData )
40         }
41         setMaxFlightHeight(flightController)
42         setMaxFlightRadius(flightController)

```

```
43     waypointMissionOperator.addListener(waypointMissionOperatorListener)
44 }
45
46 fun setMaxFlightHeight(flightController: FlightController) { //...}
47 fun setMaxFlightRadius(flightController: FlightController) { //...}
48
49 fun destroy() {
50     commandLiveData.removeObserver(commandObserver)
51     waypointMissionOperator.removeListener(waypointMissionOperatorListener)
52     statsHandler.status.removeObserver(statusObserver)
53     if (GlobalConfig.SIMULATOR_MODE) {
54         flightController.simulator.stop( //... feedback flow )
55     }
56 }
57 }
```

Źródło: Badania własne.

IV.4. Interfejs użytkownika

TODO

Rozdział V. Testy systemu oraz prezentacja użycia na wybranym case study

W poniższym rozdziale zaprezentowano, omówiono i przeprowadzono testy systemu, które świadczą o jego poprawnym działaniu.

V.1. Testy na platformie Andorid

W kontekście testowania oprogramowania na platformie Android można wyróżnić dwa rodzaje testów:

- **androidTest** - Testy, które są uruchamiane na rzeczywistych lub wirtualnych urządzeniach z systemem Android. Obejmują one m.in. test integracyjne. Szczególnie takie, w których sama JVM nie może sprawdzić działania kodu. W ramach tego rodzaju testów przetestowano serwisy odpowiedzialne za komunikację za pomocą MQTT. Konieczność ta jest spowodowana tym, że są to operacje asynchroniczne, które do swojego wykonania korzystają z narzędzi do zarządzania cyklem życia obiektów dostarczanych przez syTechnologia i producenci BSPstem Android, dlatego muszą być wykonane w ramach tego kontekstu.
- **test** - Klasyczne testy jednostkowe, które mogą być uruchamiane na lokalnej maszynie JVM. W ramach tych testów przetestowano m.in. mapowanie obiektów JSON na klasy wykorzystywane do wykonywania poszczególnych komend (klasa *pl.edu.wat.droman.data.model.command.Command*) na urządzeniu.

V.2. Testy jednostkowe

W ramach testów jednostkowych przetestowano klasę *CommandFactory*, która odpowiada za generowanie obiektów klas komend, na podstawie wartości tekstowej w formacie JSON, które są w późniejszym etapie wykonywane za pomocą klasy *CommandHandler*.

V.2.1. Test tworzenia komendy *TakeOffCommand*, za pomocą *CommandFactory*

Poniższy test (Kod. 11.) sprawdza trzy warunki:

- Czy wygenerowana komenda przez *CommandFactory* faktycznie istnieje, a nie jest przypadkiem wartością *null*?
- Czy klasa wygenerowanego obiektu jest klasą *TakeOffCommand*?
- Czy wartość parametru *type* w obiekcie jest równa stałej *TakeOffCommand.type*?

Kod. 11. Test tworzenia komendy *TakeOffCommand* za pomocą *CommandFactory*

```

1  @Test
2  fun testMappingToTakeOffCommand() {
3      //given
4      val commandFactory = CommandFactory()
5      val missionValue = "{\"type\":\"take_off\"}"
6      //then
7      val command = commandFactory.from(missionValue)
8
9      //expect
10     assertNotNull(command)
11     assertEquals(TakeOffCommand::class.java, command.javaClass)
12     assertEquals(TakeOffCommand.type, command.type)
13 }
```

Źródło: Badania własne.

V.2.2. Test tworzenia komendy *UploadWaypointCommand*, za pomocą *CommandFactory*

Poniższy przykładowy test (Kod. 12.) , w którym niestety nie można przetestować bezpośrednio wartości m.in. współrzędnych, ponieważ dostęp do nich jest chroniony przez mofikatoru dostępu, a biblioteka JUnit4 nie umożliwia dostępu do obiektów chronionych. Warto dodać, że inne frameworki, jak np. Spock, już takie rzeczy umożliwiają. Sam test służy do upewnienia się, że wartość tekstowa została zmapowana na poprawny obiekt, a w trakcie jego tworzenia nie został wywołany żaden wyjątek.

Kod. 12. Test tworzenia komendy *UploadWaypointCommand*

```

1  @Test
2  fun testMappingToUploadMissionCommand() {
3      //given
4      val commandFactory = CommandFactory()
5      val missionValue = "{\n" +
6          "        \"type\": \"upload_waypoint_mission\", \n" +
7          "        \"finished_action\": \"NO_ACTION\", \n" +
8          "        \"auto_flight_speed\": 0.01, \n" +
9          "        \"max_flight_speed\": 0.5, \n" +
10         "        \"heading_mode\": \"AUTO\", \n" +
11         "        \"waypoints\": [\n" +
12         "            {\n" +
13         "                \"attitude\": 1.0, \n" +
14         "                \"longitude\": 1.0, \n" +
15         "                \"latitude\": 1.0\n" +
16         "            }, \n" +
17         "            {\n" +
18         "                \"attitude\": 1.0, \n" +
19         "                \"longitude\": 1.0, \n" +
20         "                \"latitude\": 1.0\n" +
21         "            }, \n" +
22         "            {\n" +
23         "                \"attitude\": 1.0, \n" +
24         "                \"longitude\": 1.0, \n" +
25         "                \"latitude\": 1.0\n" +
26         "            }\n" +
27         "        ]\n" +
28         "    }"
29
30     //then
31     val command = commandFactory.from(missionValue)
32
33     //expect
34     assertNotNull(command)
35     assertEquals(UploadWaypointCommand.type, command.type)
36     assertEquals(UploadWaypointCommand::class.java, command.javaClass)
37 }

```

Źródło: Badania własne.

V.2.3. Test tworzenia komendy *UnrecognizedCommand* w przypadku nierozpoznania wartości za pomocą *CommandFactory*

Jeżeli z jakiego powodu *CommandFactory* nie będzie mógł zmapować wartości tekstowej, bo np. zostanie podany zły typ zwróci on obiekt *UnrecognizedCommand*. Niezwalająca wartości *null*, ponieważ jest to uznawane za złą praktykę, a język programowania

Kotlin wręcz do tego przymusza. Poniższy test (Kod. 13.) sprawdza wspomniany przypadek.

Kod. 13. Test tworzenia komendy *UnrecognizedCommand*

```

1  @Test
2  fun testMappingToFailUploadMissionCommand() {
3      //given
4      val commandFactory = CommandFactory()
5      val missionValue = "{\"type\": \"dsadas\"}"
6      //then
7      val command = commandFactory.from(missionValue)
8
9      //expect
10     assertNotNull(command)
11     assertEquals(UnrecognizedCommand::class.java, command.javaClass)
12     assertEquals(UnrecognizedCommand.type, command.type)
13 }

```

Źródło: Badania własne.

V.3. Testy na środowisku uruchomieniowym

W ramach testów przeprowadzono również wcześniej wspomniane testy na środowisku Android. Ich celem było przetestowanie komunikacji z brokerem MQTT za pomocą zaimplementowanych klas *MqttService*, *MqttRepository* i *ReceiveServiceTest*

V.3.1. Testy klasy *MqttRepository*

Przed uruchomieniem pojedynczych testów, JUnit uruchamia metodę z adnotacją *@Before* (Kod. 14.). W tym przypadku metoda ta ustawia zmienne w klasie dotyczące kontekstu Androida i danych logowania do brokera MQTT. Warto zaznaczyć, że dane logowania nie są zapisane nigdzie w repozytorium, a są wczytywane na etapie budowania aplikacji z pliku *gradle.properties*, który znajduje się w folderze na środowisku, na którym jest budowana aplikacja.

Poniżej przedstawiono dwa przykładowe testy. Pierwszy z nich (Kod. 15.) publikuje na wybranym Topic-u losową wartość tekstową. W asercjach sprawdzane jest czy obiekt *Result* jest uznana za sukces i czy opublikowaną wiadomość z *IMqttDeliveryToken* jest równa wygenerowanej wiadomości.

Drugi test (Kod. 16.) sprawdza, czy obiekt *Result* jest uznany za niepowodzenie, jeżeli wiadomość zostanie opublikowana na nieistniejący broker MQTT.

Kod. 14. Inicjalizowanie wartosci dla wszystkich testów w *MqttRepositoryTest*

```

1  @Before
2  fun init() {
3      appContext = InstrumentationRegistry.getInstrumentation().targetContext
4      val metadata: Bundle = appContext.packageManager.getApplicationInfo(
5          appContext.packageName,
6          PackageManager.GET_META_DATA
7      ).metadata
8
9      password = metadata.getString("mosquitto.password")!!
10     user = metadata.getString("mosquitto.user")!!
11     uri = "tcp://" + metadata.getString("mosquitto.ip")
12 }

```

Źródło: Badania własne.

Kod. 15. Test publikowania losowej wiadomości MQTT

```

1  @Test
2  fun publishExampleData() = runBlocking {
3      //given
4      val mqttRepository = MqttRepository(
5          context = appContext,
6          mqttCredentials = MqttCredentials(uri, clientId, user, password)
7      );
8      val message = "message:" + UUID.randomUUID()
9
10     //then
11     val res = mqttRepository.publish(MqttDto(ETopic.TEST, message))
12
13     //except
14     assertTrue(res.isSuccess)
15     assertEquals(message, res.getOrThrow().message.toString())
16 }

```

Źródło: Badania własne.

Kod. 16. Test publikowania wiadomości na nieistniejący broker MQTT

```

1  @Test
2  fun publishWithFailureExampleData() = runBlocking {
3      //given
4      val mqttRepository = MqttRepository(
5          context = appContext,
6          mqttCredentials = MqttCredentials("tcp://192.168.1.13", clientId, user,
7          password)
8      );
9      val message = "message:" + UUID.randomUUID()
10
11     //then
12     val res = mqttRepository.publish(MqttDto(ETopic.TEST, message))
13
14     //except
15     assertTrue(res.isFailure)
16 }

```

Źródło: Badania własne.

V.3.2. Testowanie klasy *MqttService*

Klasa *MqttService* zarządza Topic-ami MQTT. Na podstawie podanej ścieżki dla transmisji MQTT, dostarcza ona łatwy w obsłudze abstrakt (klasa *MqttService.Topic*). Można na nim wykonywać opcje publikowania i subskrybowania wiadomości. Subskrypcja polega na przekazywaniu danych przez obiekt *LiveData* z systemu Android.

Inicjowanie zmiennych w tej klasie odbywa się w taki sam sposób jak w *MqttRepositoryTest* (Kod. 14.)

Przykładowy test (Kod. 17.) sprawdza pobieranie czy pobrana wiadomość za pomocą klasy *Topic* jest równa wcześniej zapisanej wcześniej na tym Topicu. W linii nr 8 pobierana jest instancja klasy *Topic* o ścieżce *test*. Następnie w liniach 8-9 na pobrany Topic włączana jest jego subskrypcja i publikowana jest wiadomość o losowej treści. Za pomocą rozszerzenia metody *getOrAwaitValue* w języku Kotlin (ang. Extension Function) pobierana jest wartość a obiektu klasy *LiveData*. Rozszerzenie to czeka przez 5 s., aż do podanego obiektu zostanie zapisana nowa dana, albo rzuca wyjątek. W ten sposób zmienna *result* przyjmuje wartość typu *String* a nie *LiveData<String>*.

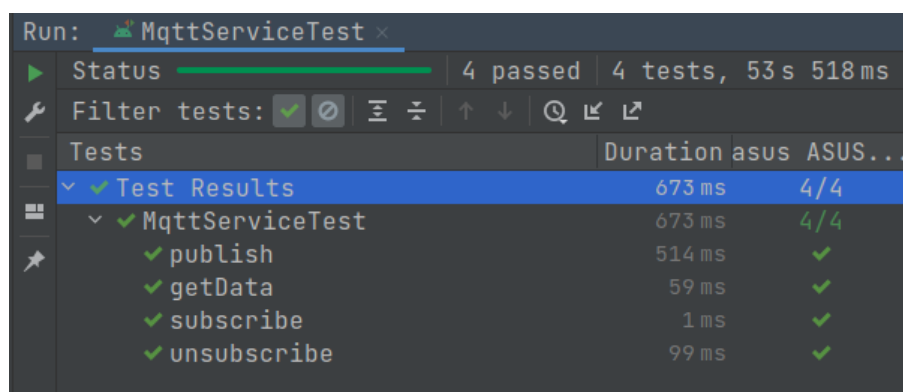
Kod. 17. Test pobierania danych za pomocą *MqttService*

```

1  @Test
2  fun getData() {
3      //given
4      val message = "message:" + UUID.randomUUID()
5      val topicVal = "test"
6
7      //then
8      val topic = mqttService.getTopic(topicVal)
9      runBlocking {
10         topic.subscribe()
11         topic.publish(message)
12     }
13     val result = topic.getData().getOrAwaitValue(time = 5).toString()
14
15     //except
16     Assert.assertEquals(message, result)
17 }

```

Źródło: Badania własne.



Tests	Duration	asus	ASUS...
✓ Test Results	673 ms	4/4	
✓ MqttServiceTest	673 ms	4/4	
✓ publish	514 ms	✓	
✓ getData	59 ms	✓	
✓ subscribe	1 ms	✓	
✓ unsubscribe	99 ms	✓	

Rys. 25. Wynik testów *MqttRepository*

Źródło: Własne

V.3.3. Testy klasy *ReceiveService*

Ostatni przedstawiony test (Kod. 18.) dotyczy klasy *ReceiveService*. W linii nr 7 na wskazany Topic, który jest przeznaczony do przesyłania komend, publikowana jest po-

lecenie zrobienia zdjęcia w formacie JSON. *ReceiveService* nasłuchuje na wspomnianym Topic-u i zwraca dane za pomocą klasy *LiveData*. Dostęp do tego obiektu uzyskuje się w linii nr 8, za pomocą metody *getCommand*. Następnie ponownie za pomocą *getOrAwaitValue* z klasy *LiveData* wyciągany jest obiekt klasy *Command*. Test polega na porównaniu parametru *type* z uzyskanego obiektu z wartością *ShootPhotoCommand.type*

Kod. 18. Test pobierania komend przez *ReceiveService*

```

1  @Test
2  fun getMission() = runBlocking {
3      //given
4      val message = "{\"type\":\"" + ShootPhotoCommand.type + "\"}"
5
6      //then
7      commandTopic.publish(message)
8      val res = receiveService.getCommand().getOrAwaitValue(time = 5)
9
10
11     //except
12     assertEquals(ShootPhotoCommand.type, res.type)
13 }

```

Źródło: Badania własne.

V.4. Testy end-to-end

Testy end-to-end to rodzaj testów automatycznych w którym sprawdzane jest działanie całego systemu, bez znajomości jego budowy. Test jest dokonywany za pomocą wystawionego interfejsu. W ramach systemu przygotowano dwa skrypty w języku Pythonm które umożliwiają ich przeprowadzenie.

V.4.1. Publikowanie na Topic-u przeznaczonym do przesyłania komend

Przedstawiony skrypt (Kod. 19.) umożliwia w prosty sposób przesyłanie komend do urządzenia końcowego. Dodatkowo w ramach skryptu zostały udostępnione dwie sekwencje *test* i *test_with_take_off*. Pierwsza z nich wykonuje trzy następujące po sobie komendy, które odpowiadają kolejno za:

- uruchomienie silników;
- zrobienie zdjęcia;
- wyłączenie silników.

Druga z nich wykonuje dodatkowo wystartowanie, tj. uniesienie się na wysokość 1,4 metra i wylądowanie. Test oznaczony jako *test* w trakcie implementacji stanowił szybką weryfikację działania, nawet w zamkniętym pomieszczeniu.

Kod. 19. Skrypt przeznaczony do publikowania na nasłuchiwanym przez urządzenie latające Topic-u komend.

```

1  if __name__ == '__main__':
2      client = mqtt.Client()
3      client.username_pw_set(username=USERNAME, password=PASSWORD)
4      client.on_connect = on_connect
5      client.connect(MQTT_SERVER, 1883, 60)
6      allowed_args = ["test", "set_home_location", "land", "upload_waypoint_mission", "
7                      take_off", "start_motors",
8                      "stop_motors", "shoot_photo", "load_waypoint_mission", "
9                      take_off_and_land", "stop_waypoint_mission",
10                     "start_waypoint_mission", "go_home"]
11     print(allowed_args)
12     if sys.argv[1] == "test":
13         client.publish(topic=MQTT_PATH + CLIENT_ID, payload='{"type":"shoot_photo"}')
14         time.sleep(1)
15         client.publish(topic=MQTT_PATH + CLIENT_ID, payload='{"type":"start_motors"}')
16         time.sleep(3)
17         client.publish(topic=MQTT_PATH + CLIENT_ID, payload='{"type":"stop_motors"}')
18     elif sys.argv[1] == "set_home_location":
19         payload = {
20             "type": "set_home_location",
21             "longitude": 20.0,
22             "latitude": 113.0,
23         }
24         client.publish(topic=MQTT_PATH + CLIENT_ID, payload=json.dumps(payload))
25     elif sys.argv[1] == "load_waypoint_mission":
26         payload = {
27             "type": "load_waypoint_mission",
28             "finished_action": "NO_ACTION",
29             "auto_flight_speed": 1.0,
30             "max_flight_speed": 5.0,
31             "heading_mode": "AUTO",
32             "waypoints": [
33                 {
34                     "attitude": 10,
35                     "longitude": START_POINT_LONGITUDE,
36                     "latitude": START_POINT_LATITUDE + ONE_METER_OFFSET * 0.5
37                 },
38                 {
39                     "attitude": 11,
40                     "longitude": START_POINT_LONGITUDE + ONE_METER_OFFSET * 1,
41                     "latitude": START_POINT_LATITUDE - ONE_METER_OFFSET * 1
42                 },
43                 {
44                     "attitude": 12,
45                     "longitude": START_POINT_LONGITUDE + ONE_METER_OFFSET * 2,
46                     "latitude": START_POINT_LATITUDE + ONE_METER_OFFSET * 1
47                 }
48             ]
49         }
50         print(json.dumps(payload))
51         client.publish(topic=MQTT_PATH + CLIENT_ID, payload=json.dumps(payload))
52     elif sys.argv[1] == "take_off_and_land":
53         client.publish(topic=MQTT_PATH + CLIENT_ID, payload='{"type":"take_off"}')
54         time.sleep(3)
55         client.publish(topic=MQTT_PATH + CLIENT_ID, payload='{"type":"shoot_photo"}')
56         time.sleep(3)
57         client.publish(topic=MQTT_PATH + CLIENT_ID, payload='{"type":"land"}')
58     elif sys.argv[1] in allowed_args:
59         client.publish(topic=MQTT_PATH + CLIENT_ID, payload='{"type":"' + sys.argv[1] +
60             '"}')
61     else:
62         print("forbidden_command_" + sys.argv[1])

```

Źródło: Badania własne.

V.4.2. Subskrybowanie Topic-u przeznaczonego do przesyłania zdjęć

Przy każdym wywołaniu polecenia zrobienia zdjęcia na BSP, po wykonaniu jest ono przesyłane w najwyższej możliwej rozdzielczości, na dedykowany Topic. Poniższy skrypt (Kod. 20.) subskrybuje go i przy każdym otrzymaniu wiadomości zapisuje ją w folderze, w którym jest uruchomiony skrypt. Szybki podgląd zdjęcia pozwala na zweryfikowanie czy transmisja i kodowanie odbywa się prawidłowo.

Kod. 20. Skrypt nasuchujący na podanym topicu MQTT i zapisujący dane z wiadomości MQTT w formacie /textjpeg

```

1 def on_connect(_client, userdata, flags, rc):
2     print("Connected_with_result_code_" + str(rc))
3     _client.subscribe(MQTT_PATH)
4
5
6 def on_message(_client, userdata, msg):
7     f = open('output.jpeg', "wb")
8     f.write(msg.payload)
9     print("Image_Received")
10    f.close()
11
12 if __name__ == '__main__':
13     client = mqtt.Client()
14     client.username_pw_set(username=USERNAME, password=PASSWORD)
15     client.on_connect = on_connect
16     client.on_message = on_message
17     client.connect(MQTT_SERVER, 1883, 60)
18     client.loop_forever()

```

Źródło: Badania własne.

V.5. Testy manualne

TODO

Podsumowanie

Bibliografia

- [1] Sarah E. Kreps “Drony. Wprowadzenie Technologie Zastosowania” Wydawnictwo Naukowe PWN, Warszawa, 2019;
- [2] A. Kelsey “Flying Robots 101: Everthing You Need to Know about Drones”, <https://www.popsoci.com/technology/article/2013-03/drone-any-other-name/> [dostęp: 20-04-2022];
- [3] “NB-IoT vs Lora” <https://ubidots.com/blog/lorawan-vs-nb-iot/#lorawan-vs-nb-iot-a-quick-overview> [dostęp: 20-04-2022];
- [4] “machine-to-machine (M2M)” <https://www.techtarget.com/iotagenda/definition/machine-to-machine-M2M> [dostęp: 20-04-2022];
- [5] “LPWA wikipedia” <https://pl.wikipedia.org/wiki/LPWAN> [dostęp: 20-04-2022];
- [6] “LoRa Technology - An Overview, IEEE, 2018” <https://ieeexplore.ieee.org/document/8474715> [dostęp: 20-04-2022];
- [7] “Wikipedia: Yuneec International” https://en.wikipedia.org/wiki/Yuneec_International [dostęp: 25-04-2022];
- [8] “On the Performance of Narrow-band Internet of Things (NB-IoT) for Delay-tolerant Services, IEEE, 2019” <https://ieeexplore.ieee.org/document/8768871> [dostęp: 20-04-2022];
- [9] “Wikipedia: SZ DJI Technology Co., Ltd.”, <https://en.wikipedia.org/wiki/DJI> [dostęp: 20-04-2022];
- [10] “De Havilland DH-82 "Tiger Moth"("Queen Bee"), 1931”, <http://www.samolotytypolskie.pl/samoloty/782/126/De-Havilland-DH-82-Tiger-Moth-Queen-Bee> [dostęp: 22-04-2022];
- [11] “DJI market share: here’s exactly how rapidly it has grown in just a few years”, <https://www.thedronegirl.com/2018/09/18/dji-market-share/> [dostęp: 22-04-2022];
- [12] “Wojna w Ukrainie: Pasjonaci dronów namierzają rosyjskie wojska”, <https://fotoblogia.pl/17711,wojna-w-ukrainie-pasjonaci-dronow-namierzaja-rosyjskie-wojska> [dostęp: 22-04-2022];
- [13] “Tureckie drony na Ukrainie pokazały wojnę przyszłości. Bayraktar TB2 wyrządzają ogromne szkody”, <https://www.chip.pl/2022/03/tureckie-drony-w-ukrainie-pokazaly-wojne-przyszlosci-bayraktar-tb2-wyrzadzaja-ogromne-szkody/> [dostęp: 22-04-2022];
- [14] “Zaskakująca skuteczność Bayraktarów. Ekspert o rosnącej roli dronów w wojnie”, <https://www.pap.pl/aktualnosci/news%2C1129159%2Czaskakujaca-skuteczna-bayraktarow-ekspert-o-rosnacej-rol-i-dronow-w> [dostęp: 22-04-2022];

- [15] “DJI Mavic Mini SE”, <https://www.dji.com/pl/mini-se?site=brandsite&from=nav> [dostęp: 20-04-2022];
- [16] “DJI store”, <https://store.dji.com> [dostęp: 20-04-2022];
- [17] “DJI Lightbridge”, <https://www.dji.com/pl/dji-lightbridge/info> [dostęp: 20-04-2022];
- [18] “DJI Lightbridge2”, <https://www.dji.com/pl/lightbridge-2/info#specs> [dostęp: 20-04-2022];
- [19] “Wikipedia: OFDM”, <https://pl.wikipedia.org/wiki/OFDM> [dostęp: 20-04-2022];
- [20] “Wikipedia: Ryan Model 147 Lightning Bug”, https://en.wikipedia.org/wiki/Ryan_Model_147 [dostęp: 22-04-2022];
- [21] “Wikipedia: MQ-1 Predator”, https://en.wikipedia.org/wiki/General_Atomics_MQ-1_Predator [dostęp: 22-04-2022];
- [22] “Wikipedia: Baykar”, <https://en.wikipedia.org/wiki/Baykar> [dostęp: 22-04-2022];
- [23] “Wikipedia: FHSS”, <https://pl.wikipedia.org/wiki/FHSS> [dostęp: 20-04-2022];
- [24] “DJI Mavic 2 - Ocusync 2.0 What is it & What’s Compatible ? + How is it different from Lightbridge”, <https://www.youtube.com/watch?v=gfqcSv9sR0A> [dostęp: 20-04-2022];
- [25] “DJI Gogle”, https://u.cyfrowe.pl/600x0/2/7/2_732250420.png [dostęp: 20-04-2022];
- [26] “Konkurs MON na bezzałogowe systemy powietrzne, lądowe, morskie”, <https://www.wojsko-polskie.pl/wat/articles/aktualnosci-w/konkurs-mon-na-bezzałogowe-systemy-powietrzne-ladowe-i-morskie/> [dostęp: 21-04-2022];
- [27] “Wikipedia: General Atomics Aeronautical Systems”, https://en.wikipedia.org/wiki/General_Atomics_Aeronautical_Systems [dostęp: 26-04-2022];
- [28] “General Dynamics: Our history”, <https://www.gd.com/about-gd/our-history> [dostęp: 26-04-2022];
- [29] “PANSA: Ruszają regularne loty transportowe BSP”, <https://www.pansa.pl/ruszaja-regularne-loty-transportowe-bsp/> [dostęp: 28-04-2022];
- [30] “Akcja TOPR dron dostarczył koce i ogrzewacze”, <http://www.swiatdronow.pl/akcja-topr-dron-dostarczyl-koce-i-ogrzewacze/> [dostęp: 28-04-2022];

- [31] “Drones to the Rescue? How Drones are Changing the Landscape of High Mountain Rescue Efforts”, <https://snowbrains.com/drones-to-the-rescue-how-drones-are-changing-the-landscape-of-high-mountain-rescue-efforts/> [dostęp: 29-04-2022];
- [32] “Drones guide rescuers to hunter lost in North Carolina woods, officials say”, <https://www.newsobserver.com/news/state/north-carolina/article260714992.html> [dostęp: 29-04-2022];
- [33] “Wleciał dronem do wnętrza wulkanu na Islandii. Ostatnie sekundy nagrania jeżą włosy na głowie [WIDEO]”, <https://www.twojapogoda.pl/wiadomosc/2021-06-02/wlecial-dronem-do-wnetrza-wulkanu-na-islandii-ostatnie-sekundy-nagrania-jeza-wlosy-na-glowie-wideo/> [dostęp: 29-04-2022];
- [34] “It’s a bird! It’s a plane! It’s a drone that makes movies!”, <https://www.washingtonpost.com/news/the-switch/wp/2013/08/15/its-a-bird-its-a-plane-its-a-drone-that-makes-movies/> [dostęp: 29-04-2022];
- [35] “How to Build a Drone: Construct Your Drone from Scratch”, <https://www.mydronelab.com/blog/how-to-build-a-drone.html> [dostęp: 29-04-2022];
- [36] “Drone Delivery Was Supposed to be the Future. What Went Wrong?”, <https://www.youtube.com/watch?v=J-M98KLgaUU> [dostęp: 29-04-2022];
- [37] “Drony przetransportują sprzęt medyczny nad Sosnowcem. Przed nami spotkanie z mieszkańcami”, <https://sosnowiec.naszemiasto.pl/drony-przetransportuja-sprzet-medyczny-nad-sosnowcem-przed/ar/c1-8493233> [dostęp: 29-04-2022];
- [38] “SOLID czyli dobre praktyki w programowaniu obiektowym”, <https://www.samouczekprogramisty.pl/solid-czyli-dobre-praktyki-w-programowaniu-obiektowym/> [dostęp: 20-05-2022];
- [39] Robert C. Martin “Clean Code: A Handbook of Agile Software Craftsmanship” Financial Times Prentice Hall, Upper Saddle River, NJ, 2008;
 “SOLID czyli dobre praktyki w programowaniu obiektowym”, <https://www.samouczekprogramisty.pl/solid-czyli-dobre-praktyki-w-programowaniu-obiektowym/> [dostęp: 20-05-2022];

Spis rysunków

Rys. 1. Latający gołąb Archytasa z Tarentu	7
Rys. 2. <i>De Havilland Queen Bee</i> i premier Wielkiej Brytanii Winston Churchill ...	8
Rys. 3. <i>Ryan Model 147 Lightning Bug</i> umieszczony pod skrzydłem samolotu transportowego	8
Rys. 4. <i>MQ-1 Predator</i> , wyposażony w rakiety <i>AGM-114 Hellfire</i>	9
Rys. 5. Bayraktar TB2	10
Rys. 6. Dron osobowy Cezarei w trakcie testów w Stambule	12
Rys. 7. Dron <i>Farada G1</i> , za pomocą którego będzie odbywał się transport medyczny w okolicach Warszawy	14
Rys. 8. Strefy wyłączone dla dronów w okolicach Warszawy	16
Rys. 9. Kontroler lotu Pixhawk	17
Rys. 10. DJI Tello	24
Rys. 11. Pierwsza wersja gogli do FPV od DJI	25
Rys. 12. DJI OcuSync Air Unit	26
Rys. 13. DJI Phantom v2.0	26
Rys. 14. DJI RC plus	27
Rys. 15. Widmo OFDM i FHSS	27
Rys. 16. Widmo OcuSync z zaznaczoną modulacją FHSS i OFDM	28
Rys. 17. Porównanie widma OcuSync i Lightbridge	28
Rys. 18. Diagram komponentów	32
Rys. 19. Diagram klas ograniczony do dostępnych komend w ramach systemu	35
Rys. 20. Diagram klasy służącej do przechowywania stanu BSP	37
Rys. 21. Diagram klas ograniczony do serwisów działających w ramach komunikacji MQTT	39
Rys. 22. Diagram klas ograniczony do obszaru kontroli lotu	40
Rys. 23. DJI Mini 2	41
Rys. 24. Kontroler do DJI Mini 2	42
Rys. 25. Wynik testów MqttRepository	59

Spis tablic

Tab. 1. Porównanie rodzajów technologii M2M.[6]	21
Tab. 2. Porównanie technologii LoRa i NB-IoT [3]	23
Tab. 3. Wymagania pozafunkcjonalne	30

Załączniki

1. Płyta CD/DVD zawierająca:
 - a) Prezentację wyników pracy dyplomowej
 - b) Kody źródłowe oprogramowania
 - c) Biblioteki programowe niezbędne do zbudowania i uruchomienia oprogramowania
 - d) ...
2. ...