
Arkitektur og design

Autonomous Drone Control

Projektnummer: 14144

Forfattere

Navn: Jens Kuhr Jørgensen

Studienummer: 11690

Dato 17-12-2014

Underskrift: _____

Navn: Thomas Fiil Lyngholm

Studienummer: 11641

Dato 17-12-2014

Underskrift: _____

Navn: Rasmus Fredensborg Jensen

Studienummer: 11471

Dato 17-12-2014

Underskrift: _____

Vejleder

Navn: Torben Gregersen

Dato 17-12-2014

Underskrift: _____

Revisionshistorik

Version	Dato	Ændring
1.0	01-09-2014	Dokument oprettet.
1.1	24-09-2014	Google Maps tilføjet som aktør. Use case 2 omdøbt fra "Start flyvning" til "Upload af startsignal".
1.2	30-09-2014	Tilføjet to use cases for server-siden: Use case 11: Send data til klient og Use case 11: Modtag data fra klient.
1.3	27-11-2014	Tilføjet use-case for dronen: Use case 13: Hent startsignal, for bedre at kunne isolere Use case 7: Flyv til destination fra serveren.
2.0	15-12-2014	Korrekturrettertelser.

Tabel 1. Revisionshistorik.

Ordforklaring

Følgende tabel indeholder en ordliste over de forkortelser og tekniske ord, der er benyttet i rapporten, og som kræver yderligere forklaring.

Forkortelse	Betydning	Forklaring
ADC	Autonomous Drone Control	Systemet
SysML	Systems Modeling Language	Hardware-struktureringsværktøj
UML	Unified Modeling Language	Software-struktureringsværktøj
bdd	Block definition diagram	
Ibd	Internal block diagram	
GPS	Global Positioning System	Satellit baseret navigeringssystem.
3G	3. generation	3. generations kommunikationsnetværk.
ESC	Electronic Speed Control	Omformer et DC input til et tre-faset output. Output'et bruges ofte til at styre en børsteløs DC-motors hastighed og retning. ESC'ens output styres af et PWM-signal.
I2C	Inter-Integrated Circuit	En to-wire kommunikationsprotokol.
SDA	Serial Data Line	Datalinje for I2C.
SCL	Serial Clock Line	Clocklinje for I2C.
UART	Universal asynchronous receiver/transmitter	Hardware der oversætter data fra parallel til seriel.
PWM	Pulse-Width Modulation	Pulsmodulation
pC	Micro controller	
App	Applikation	Android smartphone applikation.
UC	Use case	Brugsscenario
Domænelogik	Business logic	Den styrende software i et system, der f.eks. forbinder grænsefladeklasser med domæneklasser.
UI	User Interface	Brugergrænseflade.
XML	Extensible Markup Language	Opmærkningssprog der anvendes til indpakning af data.
SQL	Extensible Markup Language	Programmeringssprog til databaser.

Tabel 2. Ordforklaringstabel (1/2).

Forkortelse	Betydning	Forklaring
HTTP	Hypertext Transfer Protocol	Applikationsprotokol
TCP	File Transfer Protocol	Kommunikationsprotokol.
IDE	Integrated Development Environment	Integreret softwareudviklingsmiljø.
RC	Radio Control	Anvendelse af radiosignaler til fjernstyring.
Throttle	Throttle	Flyveterm: Beskriver, hvor meget gas dronens motorer får.
Yaw	Yaw	Flyveterm: Beskriver dronens rotation om dens egen z-akse.
Pitch	Pitch	Flyveterm: Beskriver dronens rotation om dens egen y-akse.
Roll	Roll	Flyveterm: Beskriver dronens rotation om dens egen x-akse.
AT	ATTention	Kommandosprog til modemmer og lignende.

Tabel 3. Ordforklaringstabell (2/2).

Indholdsfortegnelse

Kapitel 1 Indledning	1
1.1 Systemoverblik	1
Kapitel 2 Hardwarearkitektur og -design	5
2.1 Indledning	5
2.2 Bdd ADC	5
2.3 Ibd ADC	9
2.4 Ibd drone	10
2.5 Ibd main control	13
2.6 Hardwareopsætning	15
Kapitel 3 Softwarearkitektur og -design	17
3.1 Indledning	17
3.1.1 N + 1 view model	17
3.1.2 Perspektiver	19
3.2 Domaenemodel	20
3.3 Logical view	22
3.3.1 Iteration 1	23
3.3.2 Iteration 2	38
3.3.3 Iteration 3	50
3.3.4 Iteration 4	61
3.4 Process view	66
3.4.1 Server	66
3.4.2 Drone	69
3.4.3 App	70
3.4.4 Runtime kommunikation for det samlede system	77
3.5 Data view	78
3.5.1 Server	78
3.5.2 App	79
3.6 Deployment View	81
3.6.1 Allokering	81
3.6.2 Beskrivelse af protokol	82
3.6.3 Beskrivelse af dataformat	83
3.6.4 Serialisering af data	84
3.6.5 App	87
3.7 Implementation view	88
3.7.1 Opsætning og installation	88
3.7.2 Yderlige softwareelementer og - klasser	105

Indledning 1

I dette dokument vil systemets arkitektur og design blive beskrevet. Beskrivelsen vil tage udgangspunkt i systemets kravspecifikationsdokument¹.

Til at nedbryde og beskrive hardwaren i systemet anvendes SysML², der vha. bl.a. bdd'er³ og ibd'er⁴ identificerer hardware samt interne og eksterne forbindelser i systemet.

Til at nedbryde og beskrive softwaren anvendes UML⁵. I UML udføres bl.a. klassediagrammer, sekvensdiagrammer, tilstandsmaskiner og pakkediagrammer til at beskrive systemets software-klasser og flow'et i use case'ene. I dette dokument anvendes UML sammen med N+1 modellen⁶.

1.1 Systemoverblik

På figur 1.1 ses en skitse af systemet. Skitsen viser systemets overordnede komponenter, og hvordan de er forbundet/kommunikerer med hinanden.

¹Se dokumentet "Kravspecifikation".

²<http://www.sysmlforum.com/>

³Block definition diagram

⁴Internal block diagram

⁵<http://www.umlforum.com/>

⁶http://en.wikipedia.org/wiki/View_model



Figur 1.1. Systemoverblik.

Systemet består således af:

- 1 Android smartphone med ADC applikation
- 2 ADC server
- 3 ADC drone

Derudover benytter systemet:

- 1 GPS satellitter
- 2 Google Maps

Android smartphone med ADC applikation

Android smartphone applikationen anvendes af brugeren til at kommunikere med dronen. Her er det muligt at indstille destinationskoordinater for dronen. Derudover er det muligt at se flight log'en for dronen samt billeder taget af dronen.

ADC server

Serveren anvendes som bindeledd mellem drone og smartphone app. Den modtager koordinater fra Android app'en og gør disse tilgængelige for dronen. Derudover modtager serveren billeder samt flight log's fra dronen, og gør disse tilgængelige for app'en.

ADC drone

Dronen henter koordinater fra serveren og flyver til denne destination. Her tages billeder, der uploades til serveren. Undervejs logges relevant data om flyvningen.

GPS satellitter

GPS-systemet anvendes af både app og drone til at bestemme deres aktuelle placering.

Google Maps

Google Maps er en webbaseret kortlægningsservice, der udbydes af Google. App'en anvender denne service til at lade brugeren vælge koordinater ud fra et digitalt kort.

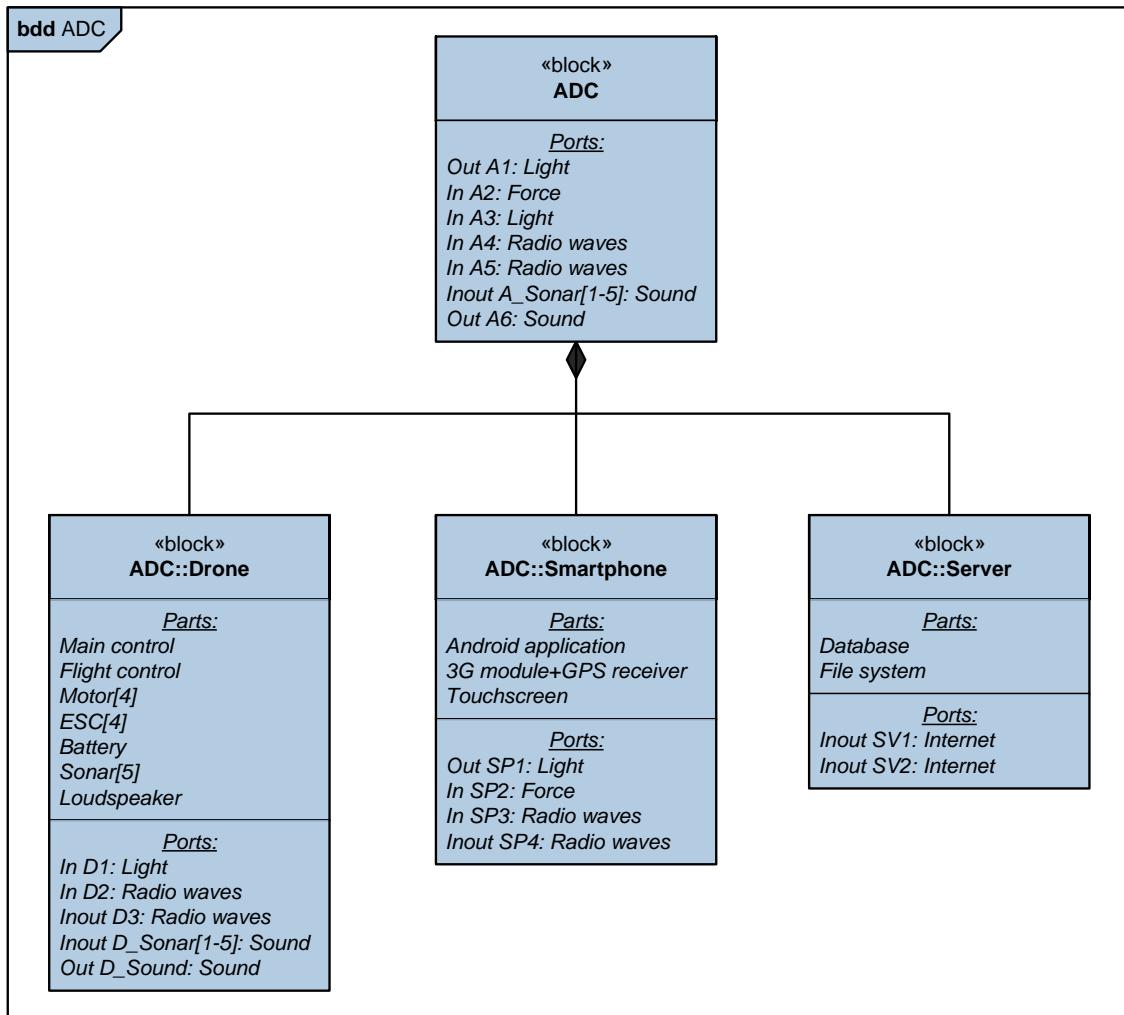
Hardwarearkitektur og -design 2

2.1 Indledning

Dette afsnit indeholder et overordnet bdd med alle hardwarekomponenter og porte i systemet. Efter bdd'et er der indsat en blokbeskrivelsestabel, der beskriver blokke og porte. Derudover indeholder afsnittet ibd'er over hhv. ADC, drone og main control. Hvert ibd er efterfulgt af en signalbeskrivelsestabel, der beskriver signaler og dertilhørende porte. Slutteligt vil afsnittet indeholde en beskrivelse af, hvordan systemets hardware forbindes.

2.2 Bdd ADC

Block definition diagram'et på figur 2.1 viser systemets overordnede enheder. På blokkene er hver bloks porte/interfaces desuden tilføjet. På diagrammet ses, at ADC har tre dele: *Drone*, *Smartphone* og *Server*, der hver har yderligere dele(parts). Portene(ports) er beskrevet med syntaksen [Retning Navn: Signaltyp]. Porten *Out A[6-10]: Sound* dækker over port *A6, A7...A10* med retning *Out* og signaltyp *Sound*. Tabel 2.1 viser bdd'ets blokbeskrivelsestabel.



Figur 2.1. Bdd for det samlede system.

Blok-navn	Funktions-beskrivelse	Port-navn	Retning	Kommentar
ADC	ADC dækker over Autonomous Drone Control. Det er det overordnede system.	A1	Out	Lys i form af skærbillede det på smartphone'en.
		A2	In	Brugerinteraktion på smartphone
		A3	In	Kameraet tager billede.
		A4	Out	Smartphones GPS modtager radiobølger.
		A5	In	Drone GPS modtager radiobølger.
		A_Sonar [1-5]	Inout	Sonaren udsender og modtager lyd.
		A6	Out	Drone signalerer takeoff.
Drone	Dronen modtager koordinater fra smartphoneren via serveren. Ved sin destination tager dronen billeder der uploades til serveren.	D1	In	Kameraet tager billeder.
		D2	In	Drone GPS modtager radiobølger.
		D3	In/out	Drone 3G modem sender og modtager data.
		D_Sonar [1-5]	In/out	Sonar udsender og modtager lyd.
		D_Sound	Out	Drone signalerer takeoff.

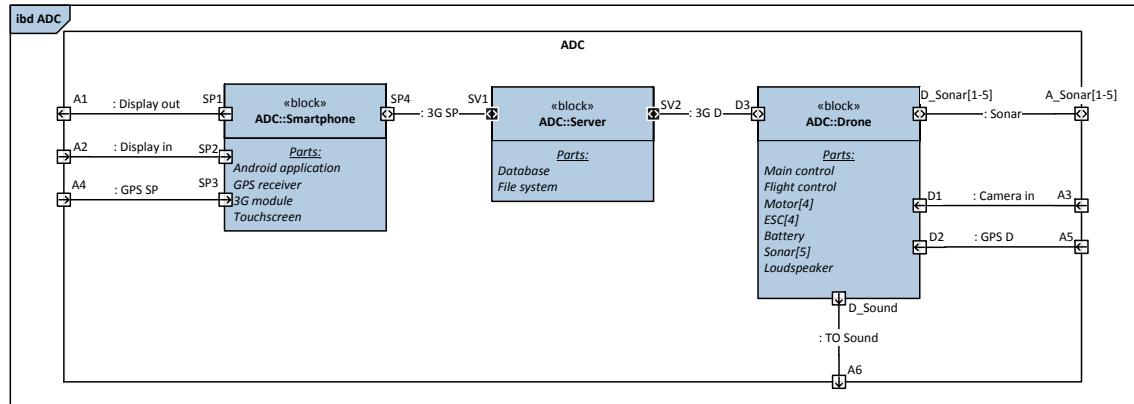
Tabel 2.1. Blokbeskrivelsestabel til ADC bdd(figur 2.1)(1/2).

Blok-navn	Funktions-beskrivelse	Port-navn	Retning	Kommentar
Smartphone	Smartphone'en udgør brugerinteraktionen i dette system. Koordinater kan bestemmes her. Derudover er det muligt at se billeder samt flight log.	SP1	Out	Lys i form af skærbillede på smartphone.
		SP2	In	Brugerinteraktion på smartphone.
		SP3	In	Smartphone's GPS modtager radiobølger.
		SP4	Inout	Smartphone's 3G modem sender og modtager radiobølger.
Server	Serveren fungerer som bindeled mellem dronen og smartphone'en. Der er ingen direkte forbindelse mellem server og drone eller server og smartphone. Forbindelsen etableres via internettet.	SV1	Inout	Server sender og modtager data via internettet
		SV2	Inout	Server sender og modtager data via internettet.

Tabel 2.2. Blokbeskrivelsestabel til ADC bdd(figur 2.1)(2/2).

2.3 Ibd ADC

På figur 2.2 ses, hvordan systemets blokke fra figur 2.1 kommunikerer internt i systemet og eksternt til omverdenen. I tabel 2.2 er signalerne mellem portene beskrevet. Pilene på portene beskriver flowets retning. De konjugerede in/out porte betyder, at det data der sendes ud fra den ene type in/out port, er input for den anden type in/out port og omvendt.



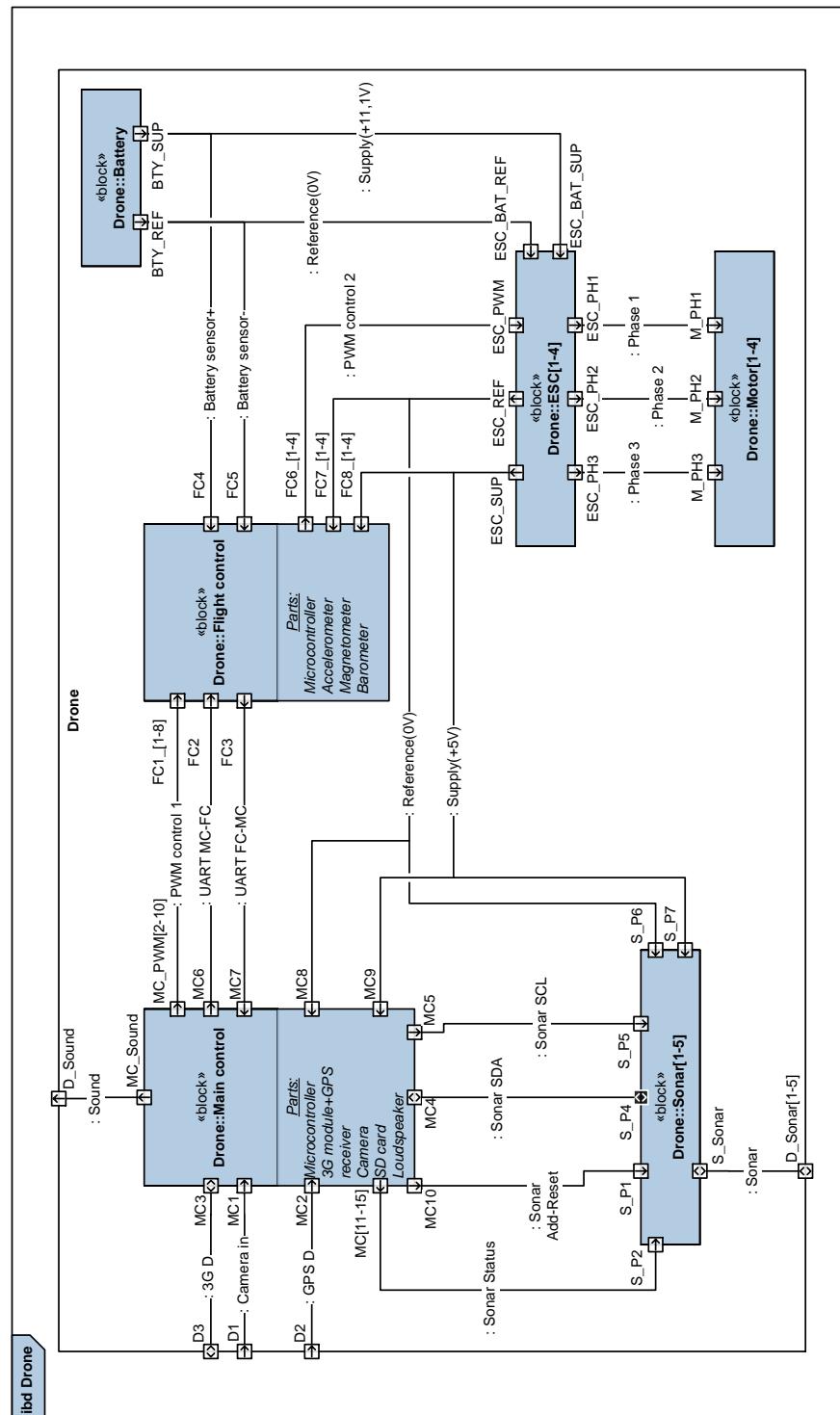
Figur 2.2. Ibd for det samlede system.

Navn	Beskrivelse	Port 1	Port 2
Display out	Lys i form af skærbillede på smartphone.	A1	SP1
Display in	Brugerinteraktion på smartphone.	A2	SP2
SP GPS	Smartphone's GPS modtager radiobølger.	A4	SP3
SV-SP	Kommunikation mellem smartphone og server. 3G modem kommunikerer til kablet internet.	SV1	SP4
SV-D	Kommunikation mellem drone og server. 3G modem kommunikerer til kablet internet.	SV2	D3
Sonar	Sonar udsender og modtager lyd.	A_Sonar [1-5]	D_Sonar [1-5]
TO Sound	Drone signaler take-off.	A6	D_Sound
Camera in	Kamera tager billeder.	A3	D1
D GPS	Drone GPS modtager radiobølger.	A5	D2

Tabel 2.3. Signalbeskrivelsestabel til ADC ibd(figur 2.2).

2.4 Ibd drone

På figur 2.3 ses den interne kommunikation i dronen. Main control modtager GPS- samt sonarsensordata. Derudover modtages data fra flight control's sensorer. Ud fra dette beregnes styringen af dronen, der sendes fra main control via flight control til ESC'erne og endeligt til motorerne. Batteriet forsyner alle blokkene, og spændingen på batteriet overvåges af flight controlleren.

*Figur 2.3.* Ibd for drone.

Navn	Beskrivelse	Port 1	Port 2
Camera in	Kameraet tager billeder.	D1	MC1
GPS D	Drone GPS modtager radiobølger.	D2	MC2
3G D	Kommunikation mellem smartphone og server.	D3	MC3
Sonar	Sonar udsender og modtager lyd. Hver af de fem sonarer har en "S_Sonar"-port og dermed et "Sonar"-signal.	D_Sonar [1-5]	S_Sonar
Sonar SDA	SDA(I2C) mellem main control og sonar. Sonarer har fælles SDA.	S_P4	MC4
Sonar SCL	SCL(I2C) mellem main control og sonar. Sonarer har fælles SCL.	S_P5	MC5
Sonar Status	Digitalt statussignal. Indikerer sonarens nuværende tilstand. Hver af de fem sonarer har en "S_P2"-port og dermed et "Sonar Status"-signal.	MC[11-15]	S_P2
Sonar Add-Reset	Digitalt signal. Main control bestemmer hvilken I2C adresse der bruges.	MC10	S_P1
UART MC-FC	TX(UART) fra main control til flight control.	MC6	FC2
UART FC-MC	TX(UART) fra flight control til main control.	MC7	FC3
Reference (0V)	Stelforbindelse (0V)	MC8 + S_P6 + FC7 [1-4]	ESC_REF
Supply (+5V)	Spændingsforsyning (+5V)	MC9 + S_P7 + FC8 [1-4]	ESC_SUP
PWM control 1	PWM styresignaler fra main control til flight control.	MC_PWM [2-10]	FC1_[1-8]
PWM control 2	PWM signal. Hver af de fire ESC'er har en "ESC_PWM"-port.	FC6_[1-4]	ESC_PWM

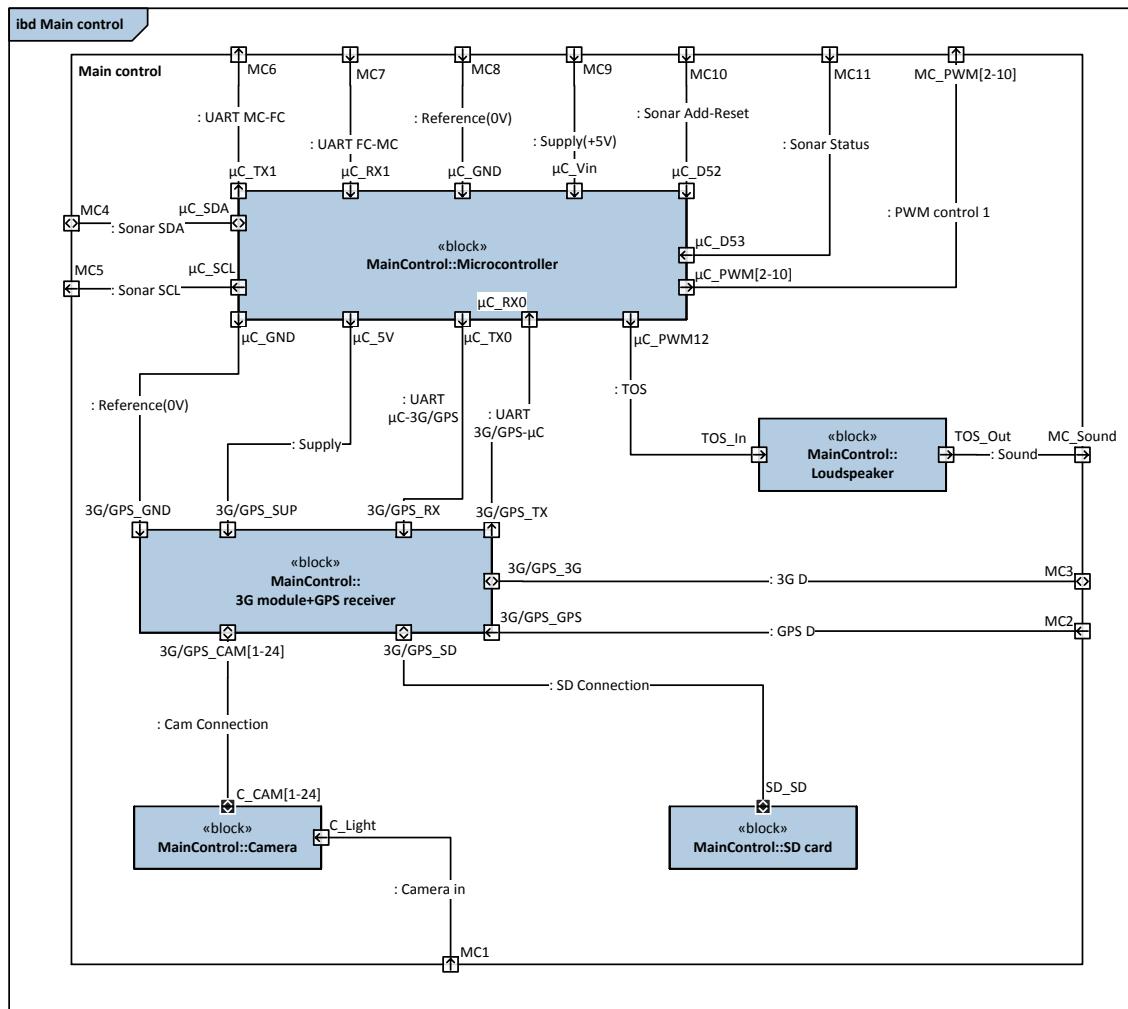
Tabel 2.4. Signalbeskrivelsestabel til drone ibd(figur 2.3)(1/2).

Navn	Beskrivelse	Port 1	Port 2
Battery sensor-	Batterisensor (0V)	FC5	BTY_REF
Battery sensor+	Batterisensor (11,1V)	FC4	BTY_SUP
Reference (0V)	Stelforbindelse (0V)	BTY_REF	ESC_BAT_REF
Supply (11,1V)	Spændingsforsyning (+11,1V)	BTY_SUP	ESC_BAT_SUP
Phase 1	Fase 1 til motor.	ESC_PH1	M_PH1
Phase 2	Fase 2 til motor.	ESC_PH2	M_PH2
Phase 3	Fase 3 til motor.	ESC_PH3	M_PH3
Sound	Drone signalerer take-off.	MC_Sound	D_Sound

Tabel 2.5. Signalbeskrivelsestabel til drone ibd(figur 2.3)(2/2).

2.5 Ibd main control

På figur 2.4 ses den interne kommunikation i blokken main control. Main control består af et 3G/GPS shield¹ monteret på et Arduino Mega2560 board. Derudover dækker main control over en højttaler, et kamera og et SD-kort. Micro controller'en tager billeder med kameraet, og gemmer dem på SD-kortet, samtidig med at de sendes til serveren via 3G-modulet. Dronens aktuelle placering aflæses fra GPS'en, og take-off varsles med højttaleren.



Figur 2.4. Ibd for main control.

¹<http://www.cooking-hacks.com/documentation/tutorials/arduino-3g-gprs-gsm-gps>

Navn	Beskrivelse	Port 1	Port 2
Camera in	Kameraet tager billeder.	MC1	C_Light
GPS D	GPS signal modtages.	MC2	3G/ GPS_GPS
3G D	3G signaler sendes og modtages.	MC3	3G/GPS_3G
Sonar SDA	SDA(I2C) mellem main control og sonar.	MC4	μC_SDA
Sonar SCL	SCL(I2C) mellem main control og sonar.	MC5	μC_SCL
UART MC-FC	TX(UART) fra main control til flight control	μC_TX1	MC6
UART FC-MC	TX(UART) fra flight control til main control	μC_RX1	MC7
Reference (0V)	Stelforbindelse (0V)	MC8	μC_GND
Supply (+5V)	Spændingsforsyning (+5V)	MC9	μC_SUP
Sonar Add-Reset	Digitalt signal. Main control bestemmer hvilken I2C adresse der bruges.	MC10	μC_D52
Sonar Status	Digitalt statussignal. Indikerer sonarens nuværende tilstand.	MC11	μC_D53
PWM control 1	PWM styresignaler fra main control til flight control.	MC_PWM [2-10]	μC_PWM [2-10]
Reference (0V)	Stelforbindelse (0V)	μC_GND	3G/ GPS_GND
Supply (+5V)	Spændingsforsyning (+5V)	μC_5V	3G/ GPS_SUP
UART μC-3G/GPS	TX(UART) fra μC til 3G/GPS	μC_TX0	3G/ GPS_RX
UART 3G/GPS-μC	TX(UART) fra 3G/GPS til μC	μC_RX0	3G/ GPS_TX
Cam Connection	Forbindelse mellem 3G/GPS og kamera.	3G/GPS _CAM[1-24]	C_CAM [1-24]
SD Connection	Forbindelse mellem 3G/GPS og SD-kort.	3G/GPS_SD	SD_SD
TO Sound	Lydstyringssignal til take-off lydsignal.	μC_PWM12	TOS_In
Sound	Lydsignal før take-off.	TOS_S_Out	MC_Sound

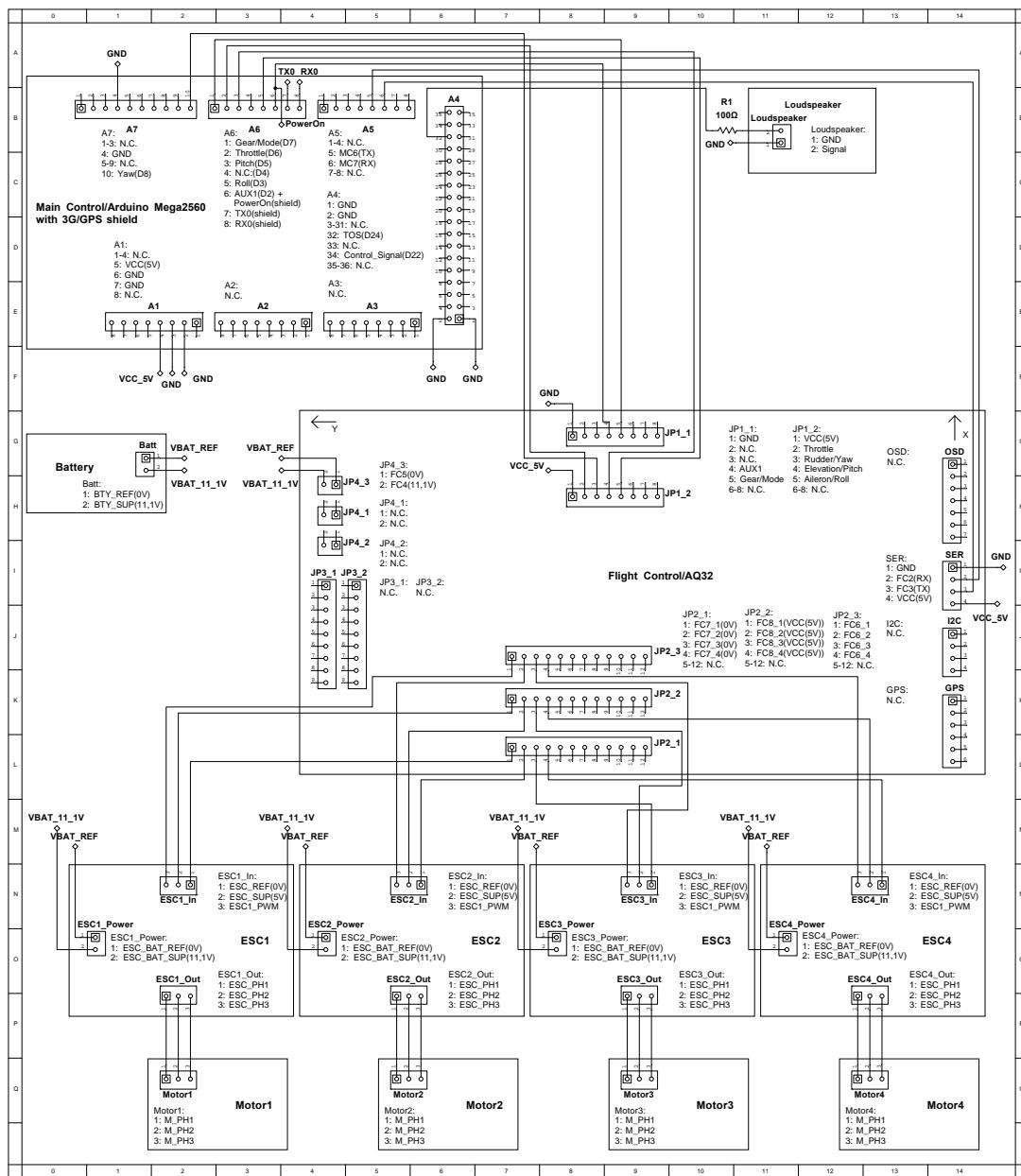
Tabel 2.6. Signalbeskrivelsestabel til main control ibd(figur 2.4).

2.6 Hardwareopsætning

I dette afsnit beskrives, hvordan det implementerede hardware i systemet forbindes. På figur 2.5 ses systemets hardwarekomponenter med de nødvendige fysiske forbindelser. De pins der tidligere er beskrevet i ibd'erne, er på figur 2.5 navngivet herefter.

Arduino'en og AeroQuad Flight Control Board'et kører på hhv. 5 og 3,3V, og det har derfor været nødvendigt at undersøge, hvorvidt spændingsbeskyttelse på AeroQuad board'et er påkrævet. Jvf. databladet for AeroQuad board'et, er alle pins anvendt i dette system 5V tolerante, og der er derfor ikke tilføjet nogen beskyttelse mellem boards'ene.

Figur 2.5 er udarbejdet i Multisim, og kan findes på projekt-CD'en i en PDF version i A2 format.



Figur 2.5. Hardwareforbindelser i det implementerede system.

Softwarearkitektur og -design 3

3.1 Indledning

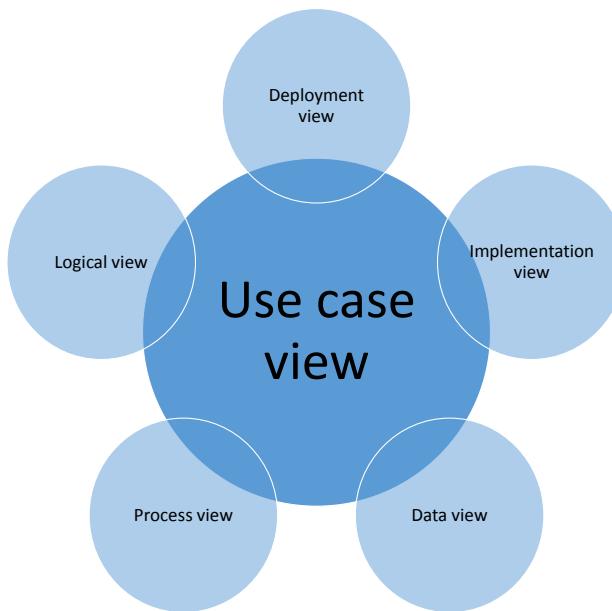
Dette afsnit indeholder en detaljeret beskrivelse af systemets softwarearkitektur og -design. Afsnittet forsøger at belyse softwaren fra alle relevante vinkler vha. diverse UML diagrammer. Der arbejdes ud fra N+1 view model'en¹.

Afsnittet beskriver use case'ene i en stigende rækkefølge, og ikke i den rækkefølge de er implementeret i systemet på. For implementeringsrækkefølgen se prioriteringstabellen i dokumentet "Kravspecifikation".

3.1.1 N + 1 view model

N + 1 view model'en bruges til at beskrive et softwaresystem for forskellige interesser med forskellige syn og interesser i systemet. Modellen tager udgangspunkt i et use case view, der repræsenteres ved "+ 1" i modellen. Derefter er det op til systemudviklerne at bestemme, hvor mange views der skal til, for at systemet er tilstrækkeligt belyst. Til dette system anvendes der en 5 + 1 view model. De anvendte views ses på figur 3.1.

¹http://en.wikipedia.org/wiki/View_model



Figur 3.1. 5 + 1 view model.

3.1.1.1 Use case view

Use case view'et består af use case diagrammer, og beskriver systemets brugsscenarier. Udover et use case view er der anvendt et eller flere views til at belyse hver use case. Use case diagrammerne kan findes i dokumentet "Kravspecifikation".

3.1.1.2 Logical view

Logical view'et består af tre typer diagrammer: Sekvensdiagrammer, tilstandsdiagrammer/-maskiner og klassediagrammer. View'et beskriver flowet i systemet og hvordan systemet internt opererer under de forskellige brugsscenarier. I dette projekt benyttes ydermere tre-lags modellen² til at beskrive app'en, hvor klassediagrammet inddeltes i tre lag:

- Presentation tier
- Logic tier
- Data tier

Præsentationslaget indeholder de UI elementer som brugeren bliver præsenteret for. Det logiske lag indeholder alle klasser der står for funktionalitet, beregninger og logiske beslutninger. Datalaget indeholder interfaces til databaser, filsystemer o.l., hvor information gemmes og hentes.

²http://en.wikipedia.org/wiki/Multitier_architecture

3.1.1.3 Process view

Process view'et beskriver de forskellige processer/tråde i systemet, hvordan samspillet imellem trådene er, hvilke processer der kører sideløbende med hinanden og hvornår.

3.1.1.4 Deployment view

Deployment view'et kaldes også nogle gange physical view, og beskriver på hvilke fysiske komponenter softwarekomponenterne er implementeret. Derudover beskrives de anvendte kommunikationsprotokoller.

3.1.1.5 Data view

Data view'et beskriver hvordan persistent data lagres i systemet. I dette view beskrives også de anvendte databasers struktur.

3.1.1.6 Implementation view

Implementation view'et kaldes også for development view, og beskriver bl.a. filstrukturen i systemet, samt hvilken compiler der er anvendt til at compilere koden. Formålet med view'et er at muliggøre en reproduktion af softwaren.

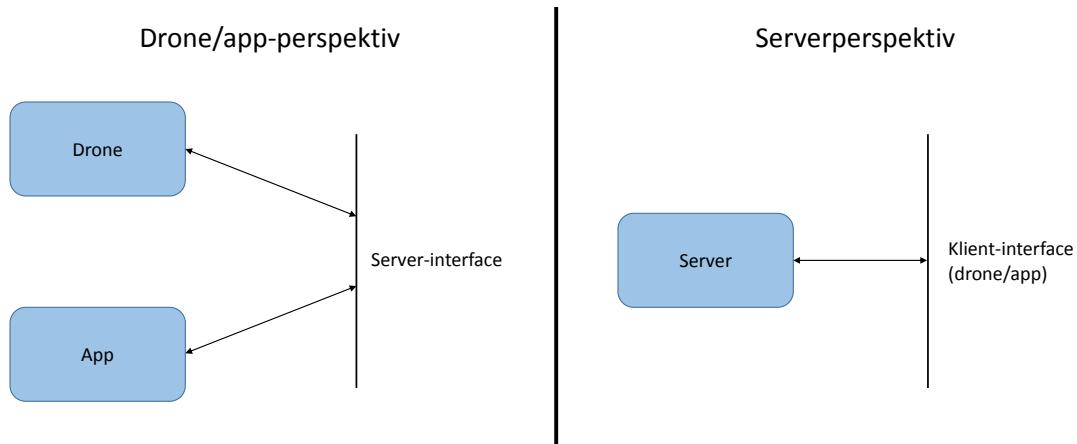
3.1.2 Perspektiver

Af designmæssige årsager skelnes der i softwarearkitekturen/-designet mellem to forskellige perspektiver: Et fra dronen og app'en og et fra serveren. Dette er gjort for at simplificere forståelsen for systemet i situationer, hvor et grundigt kendskab til det samlede system ikke er nødvendigt.

På figur 3.2 ses de to perspektiver. I drone/app-perspektivet er serverens opbygning og interne funktionalitet uden relevans. Her fokuseres blot på hvordan interfacet til serveren anvendes af dronen og app'en.

I serverperspektivet skelnes der ikke imellem kommunikation med dronen eller app'en. Serveren ser blot en klient, og er derfor kun interesseret i klientens interface.

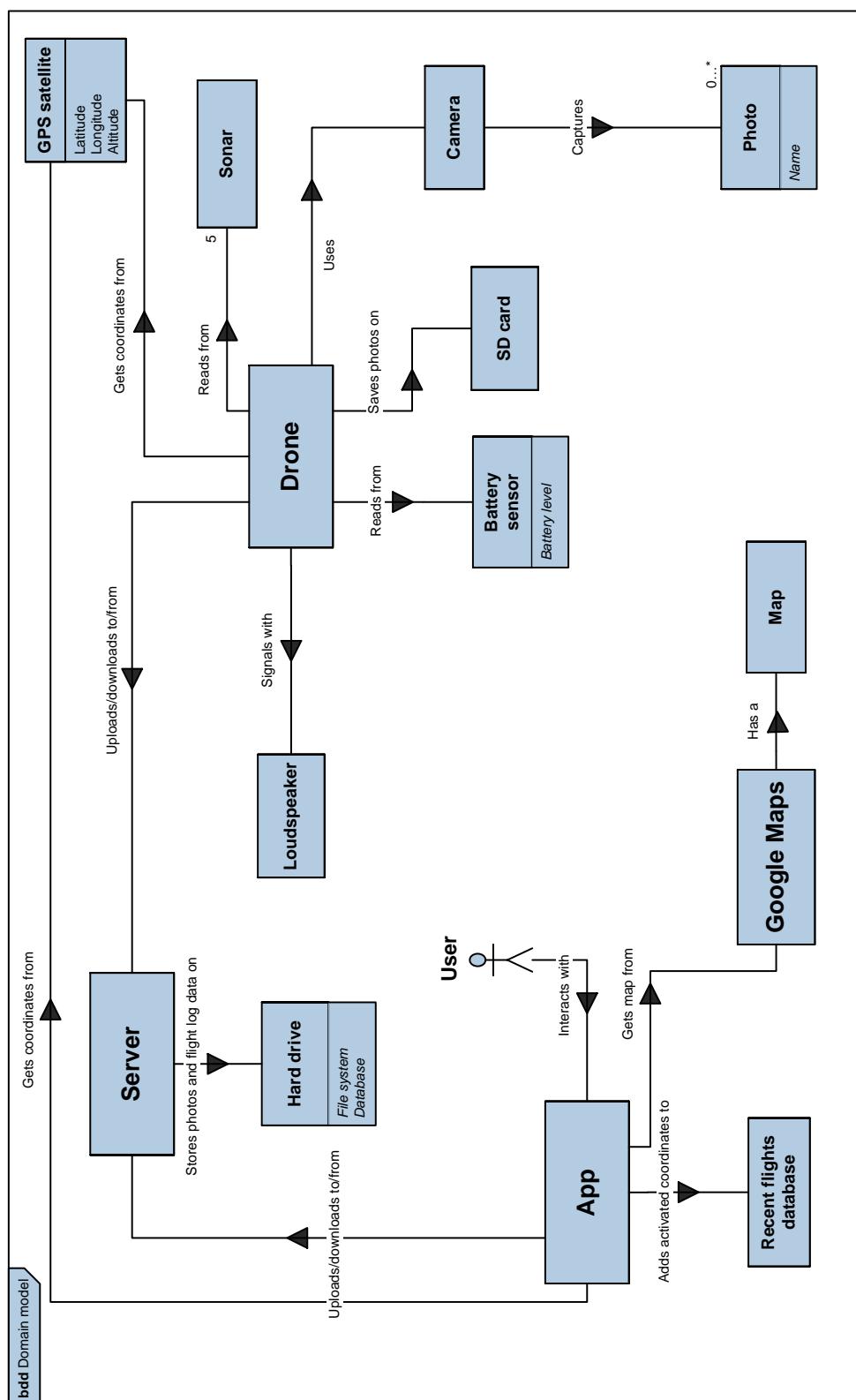
I afsnit hvor der skelnes mellem drone/app-perspektiv eller serverperspektiv, vil dette være beskrevet.



Figur 3.2. Perspektiver.

3.2 Domænemodel

Domænemodellen er et værktøj der bruges til at gå fra kravspecifikation til softwarearkitektur/-design. Modellen er udarbejdet sammen med kunden, og faciliterer overgangen fra krav til strukturering og design. Domænemodellen identificerer objekter fra use casene beskrevet som software-objekter, og beskriver software-objekternes interne relationer. Domænemodellen for systemet ses på figur 3.3.

**Figur 3.3.** Domænemodel for det samlede system.

3.3 Logical view

Logical view består af sekvensdiagrammer, tilstandsdiagrammer/-maskiner og klassediagrammer³. Til at udforme diagrammerne anvendes applikationsmodellen.

Applikationsmodel

Applikationsmodellen er et værktøj, der anvendes til at arbejde sig igennem logical view-fasen. Modellen tager udgangspunkt i use casene og domænemodellen på figur 3.3. Ud fra disse identificeres de overordnede klasser til hver use case. Herefter laves der sekvens- og tilstandsdiagrammer med de fundne klasser. Dette munder ud i klassediagrammer med fundne metoder og attributter fra den indledende analyse. Applikationsmodellen lægger op til en iterativ udviklingsprocess, hvor sekvens- og tilstandsdiagrammerne anvendes til løbende at opdatere klassediagrammerne. Logical view'et er derfor inddelt iterationsvis jf. beskrivelsen i kravspecifikationen⁴.

I applikationsmodellen identificeres tre forskellige typer klasser fra domænemodellen og usecasene:

- Boundary-klasser
- Controller-klasser
- Domain-klasser

Boundary-klasser

Boundary-klasser repræsenterer use case-aktører, og beskriver aktørernes grænseflader til systemet. Boundary-klasserne præsenterer systemet, men indeholder ingen domænelogik.

Controller-klasser

Controller-klasser indeholder systemets domænelogik, og står således for bl.a. at forbinde boundary-klasser med domain-klasser samt at håndtere bruger-input og -output.

Domain-klasser

Domain-klasser repræsenterer systemets domæne. Domain-klasserne indeholder systemets persistente elementer som f.eks. databaser, indstillinger o.l.

³http://en.wikipedia.org/wiki/4%2B1_architectural_view_model

⁴Se dokumentet "Kravspecifikation".

3.3.1 Iteration 1

I iteration 1 er use case 1, 7 og 8 implementeret. Disse use cases dækker over den mest basale funktionalitet i systemet.

3.3.1.1 Use case 1: Vælg koordinater

Use case 1 beskriver hvordan brugeren vælger de koordinater på app'en, der senere skal sendes til serveren og videre til dronen. Brugeren har mulighed for at vælge koordinaterne på fire forskellige måder: Via Google Maps, tidligere flyvninger, brugerens lokation eller ved manuel indtastning. Use casen beskriver et rent Android-brugsscenario, uden indflydelse fra server og drone. Ud fra use casen og domænemodellen er klasserne på figur 3.4 identificeret.



Figur 3.4. Fundne klasser for use case 1.

Klassebeskrivelser

UI: *UI* er en boundary-klasse, der indeholder de grafiske elementer i app'en, og bestemmer således app'ens fremtoning over for brugeren. *UI*-klassen indeholder ingen funktionalitet. I dette system er det meste af *UI*'et implementeret statisk, der for Java Android betyder, at det er implementeret som xml-filer. *UI*'et har et stort ansvarsområde, og implementeres derfor som flere xml-filer. *UI* er således ikke en klasse, men en pakke bestående af flere grafiske klasser. Klasserne i pakken vil senere blive udspecifieret.

ChooseCoordinates: *ChooseCoordinates* er en controller-klasse, der står for at koordinere al kommunikation og funktionalitet i app'en. Klassen er en aktiv klasse, og inddeltes pga. dens store ansvarsområde også i flere klasser. *ChooseCoordinates* er således ikke en klasse, men en pakke bestående af flere aktive klasser. Klasserne i pakken vil senere blive udspecifieret.

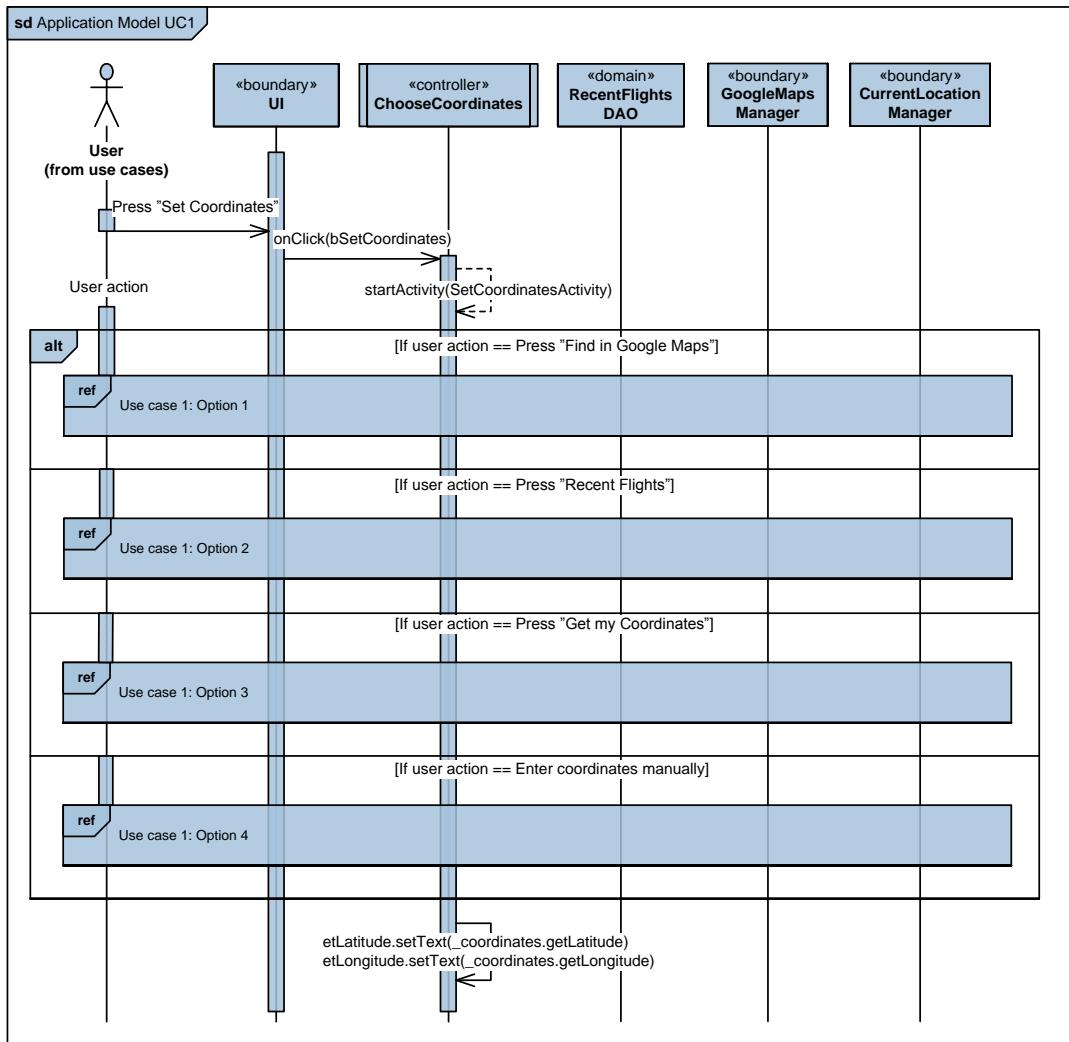
RecentFlightsDAO: *RecentFlightsDAO* er en domænekasse, der administrerer app'ens database. Databasen indeholder tider og koordinater for alle tidligere flyvninger.

GoogleMapsManager: *GoogleMapsManager* er en boundary-klasse, der står for app'ens kommunikation med Google Maps servicen.

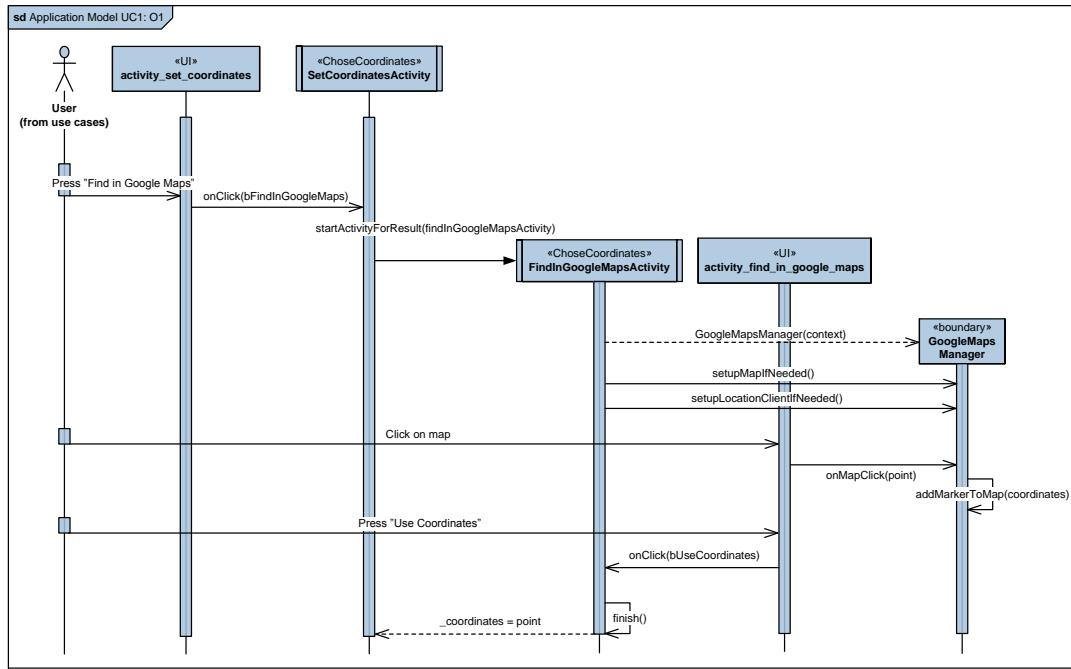
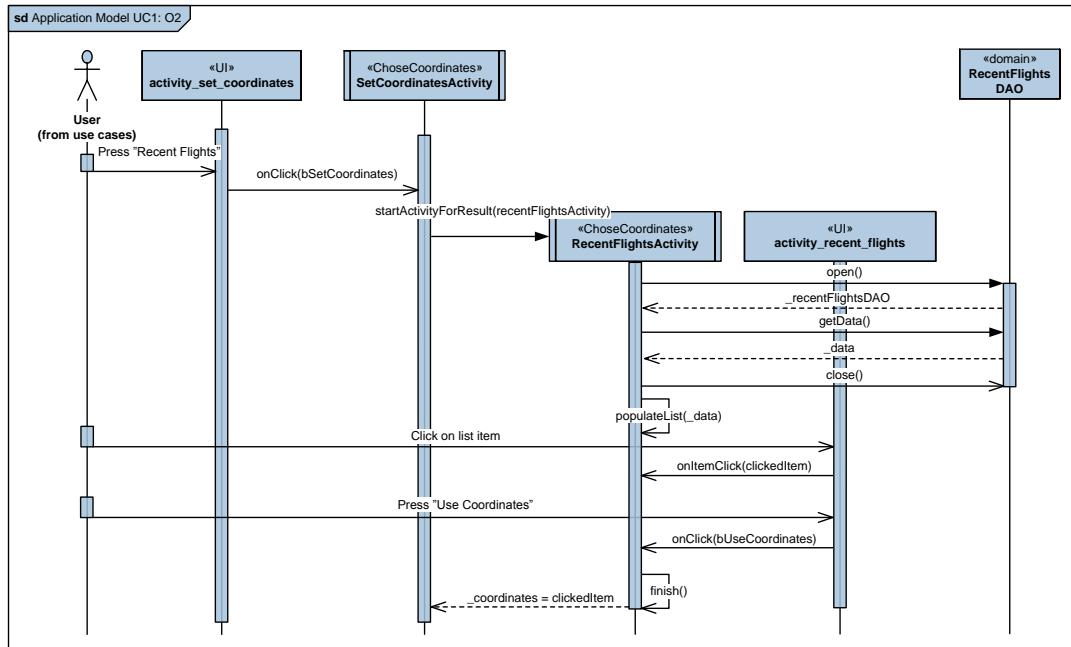
CurrentLocationManager: *CurrentLocationManager* er en boundary-klasse, der står for app'ens kommunikation med GPS-satellitterne.

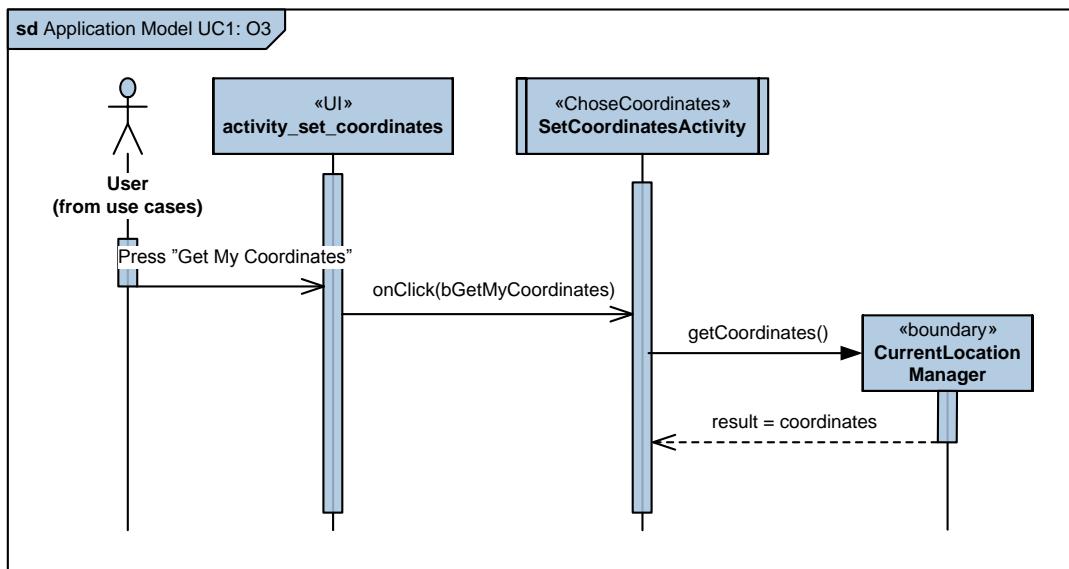
Sekvensdiagram

Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen. Derudover identificerer sekvensdiagrammet nogle af de essentielle metoder og attributter for klasserne. På figur 3.5 ses det overordnede sekvensdiagram for use casen. De fire forskellige måder at vælge koordinater på er efterfølgende beskrevet med sekvensdiagrammerne på figur 3.6 til 3.9.



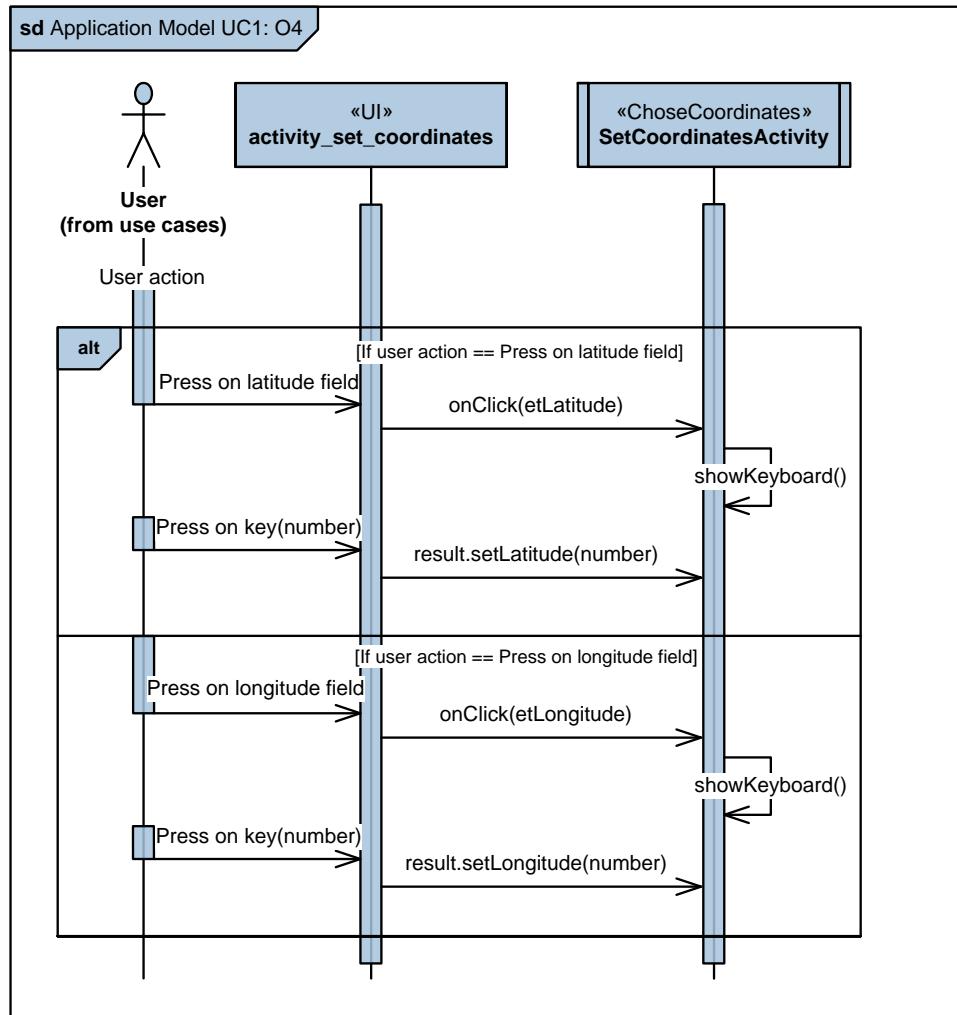
Figur 3.5. Sekvensdiagram for use case 1.

O1: "Find in Google Maps"*Figur 3.6.* Sekvensdiagram for use case 1: Option 1.**O2: "Recent Flights"***Figur 3.7.* Sekvensdiagram for use case 1: Option 2.

O3: "Get my Coordinates"

Figur 3.8. Sekvensdiagram for use case 1: Option 3.

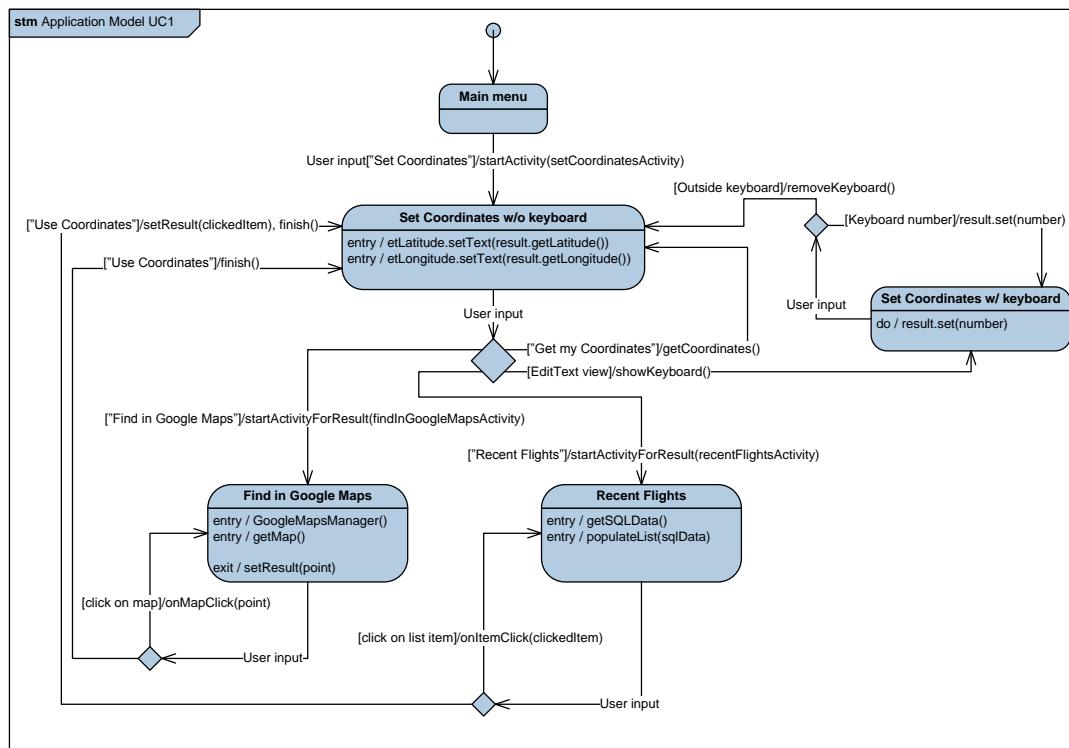
O4: Manuel indtastning



Figur 3.9. Sekvensdiagram for use case 1: Option 4.

Tilstandsdiagram

På figur 3.10 ses tilstandsdiagrammet for use case 1. Diagrammet består af fem forskellige tilstande, der alle er repræsenteret ved en Android activity. Pilene viser, hvordan brugeren har mulighed for at komme fra en tilstand til en anden.



Figur 3.10. Tilstandsmaskine for use case 1.

3.3.1.2 Use case 7: Flyv til destination

Domænemodellen er analyseret jf. use case 7 "Flyv til destination". På figur 3.11 ses de fundne klasser.



Figur 3.11. Fundne klasser for use case 7.

Klassebeskrivelser

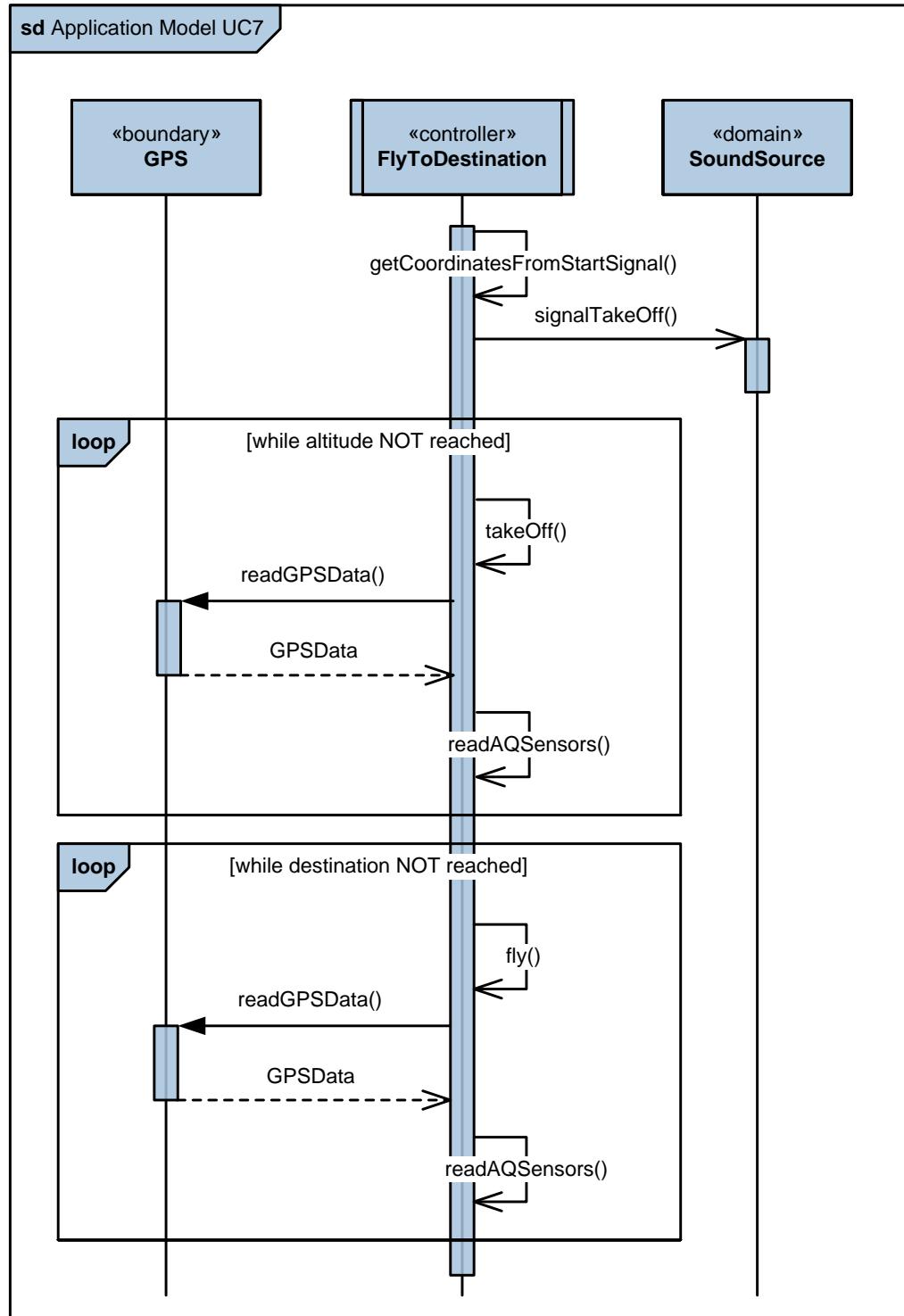
GPS: GPS-klassen er en boundary-klasse, der står for Arduino's kommunikation med GPS-satellitterne. Klassen indeholder metoder til at modtage lokationsdata.

FlyToDestination: FlyToDestination er en controller-klasse, der står for at koordinere al funktionalitet og kommunikation på Arduino'en.

SoundSource: SoundSource er en boundary-klasse. Den står for kommunikation til de eksterne lydgivere som anvendes ved take-off.

Sekvensdiagram

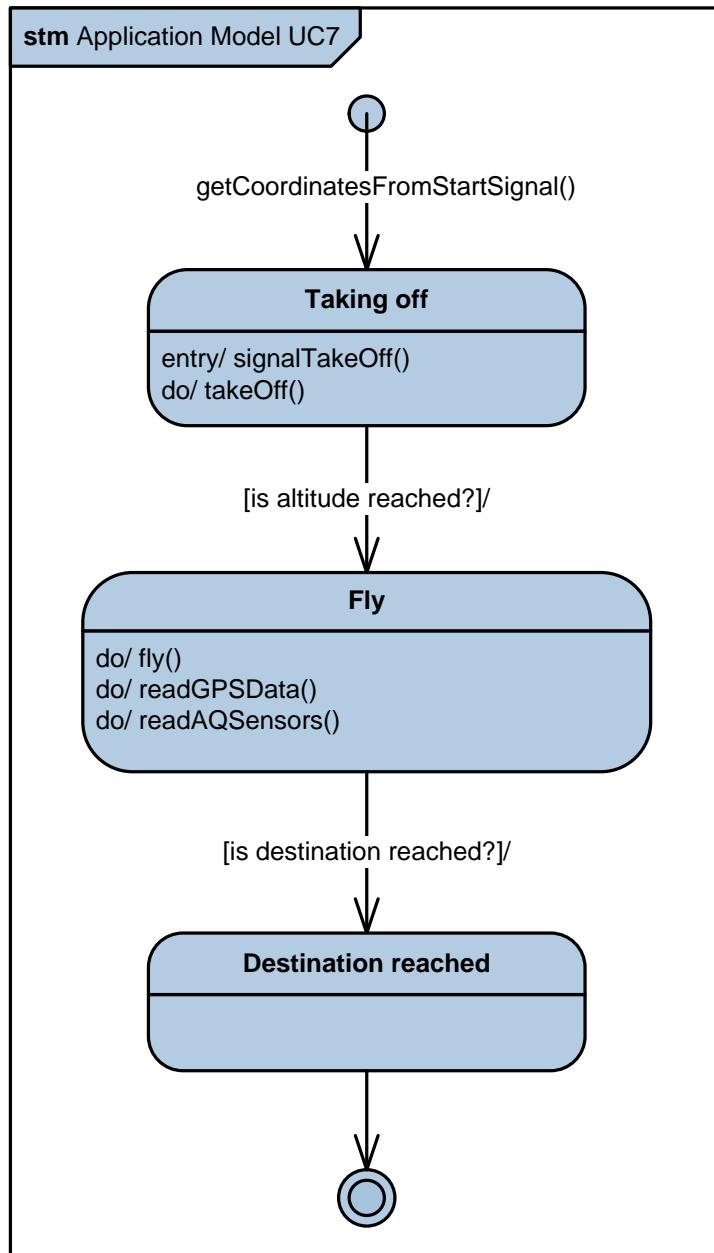
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver, hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen.



Figur 3.12. Sekvensdiagram for use case 7.

Tilstandsdiagram

På figur 3.13 ses tilstandsdiagrammet for use case 7. Diagrammet beskriver, hvordan og hvornår dronen bevæger sig fra en tilstand til en anden under udførslen af use casen, samt hvilke væsentlige opgaver dronen udfører i de givne tilstade.



Figur 3.13. Tilstandsdiagram for use case 7.

3.3.1.3 Use case 8: Flyv tilbage

Domænemodellen er analyseret jf. use case 8 "Flyv tilbage". På figur 3.14 ses de fundne klasser.



Figur 3.14. Fundne klasser for use case 8.

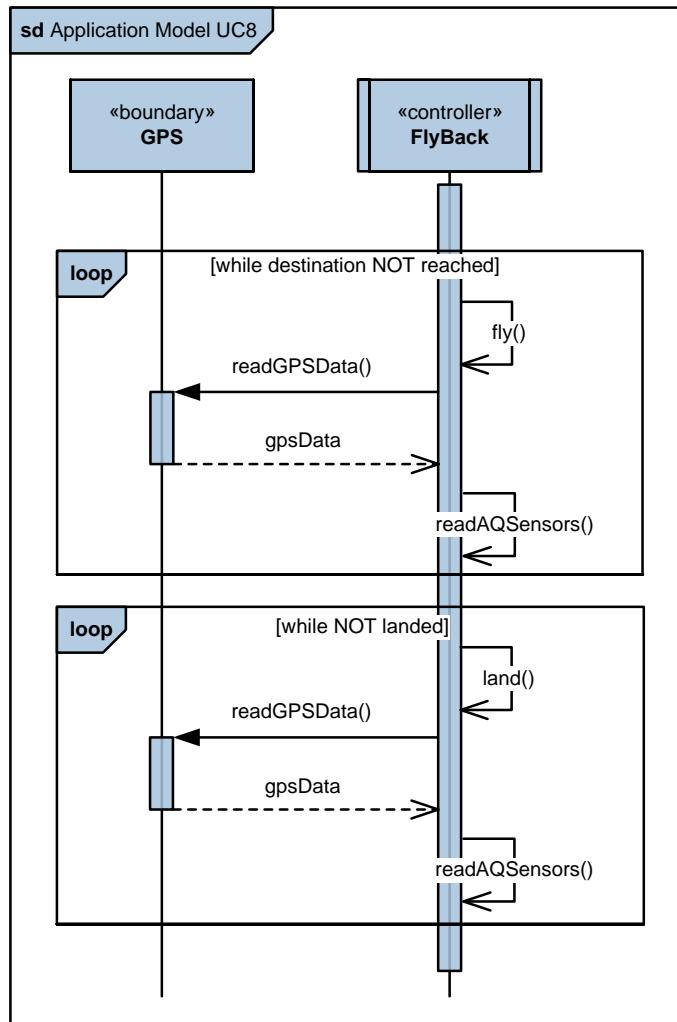
Klassebeskrivelser

GPS: *GPS*-klassen er en boundary-klasse, der står for Arduino'ens kommunikation med GPS-satellitterne. Klassen indeholder metoder til at modtage lokationsdata.

FlyBack: *FlyBack* er en controller-klasse, der står for at koordinere al funktionalitet og kommunikation på Arduino'en.

Sekvensdiagram

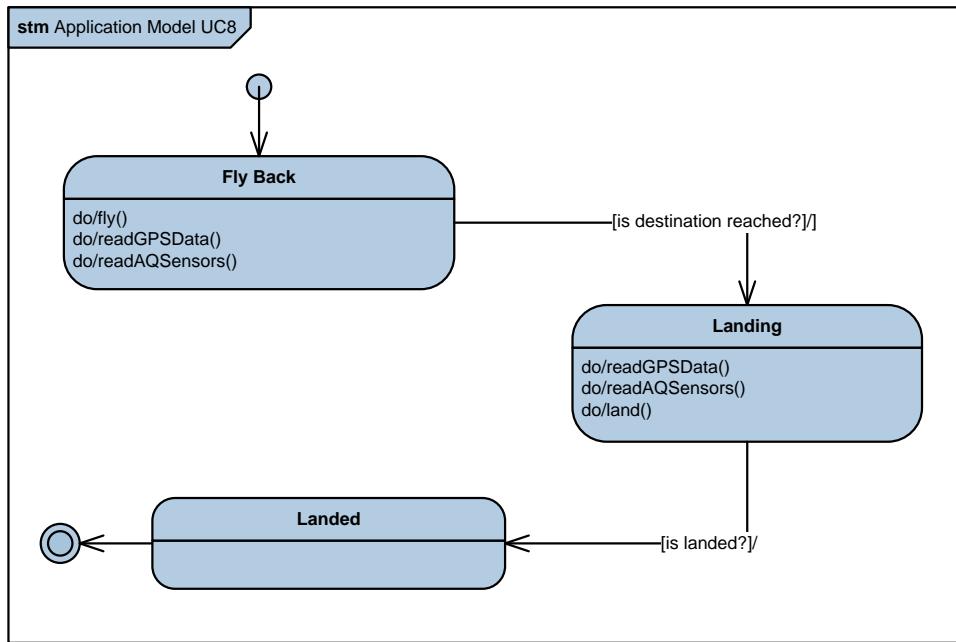
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver, hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen.



Figur 3.15. Sekvensdiagram for use case 8.

Tilstandsdiagram

På figur 3.16 ses tilstandsdiagrammet for use case 8. Diagrammet beskriver, hvordan og hvornår dronen bevæger sig fra en tilstand til en anden under udførslen af use casen, samt hvilke væsentlige opgaver dronen udfører i de givne tilstade.



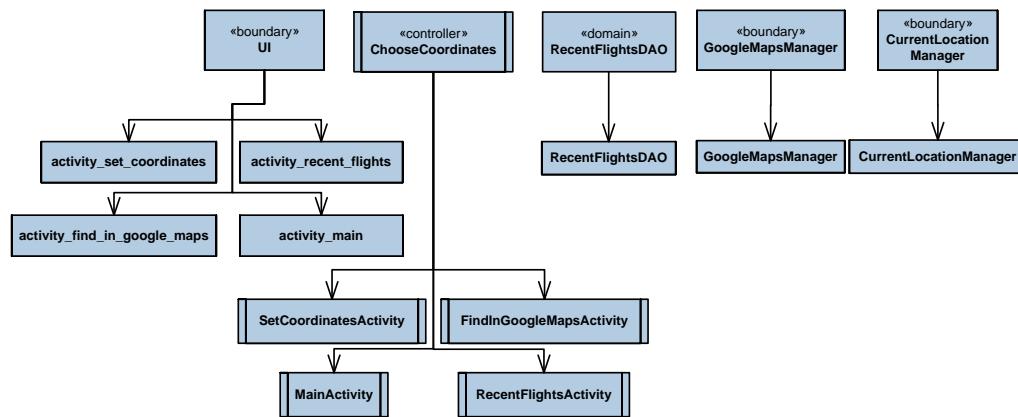
Figur 3.16. Tilstandsdiagram for use case 8.

3.3.1.4 Klassediagrammer

Ved analyse af applikationsmodellerne for use case 1, 7 og 8, resulterer iteration 1 i klassediagrammerne beskrevet i det følgende afsnit. Diagrammerne dækker over de primære klasser, metoder og attributter, der er fundet under analysen. Hver klasse, der er identificeret i applikationsmodellen, resulterer i softwareklasser. For de tre systemdele app, drone og server, vises først, hvordan hver applikationsklasse udpakkes til en eller flere softwareklasser, der varetager applikationsklassens ansvar. Derefter vises klassediagrammet med disse softwareklasser.

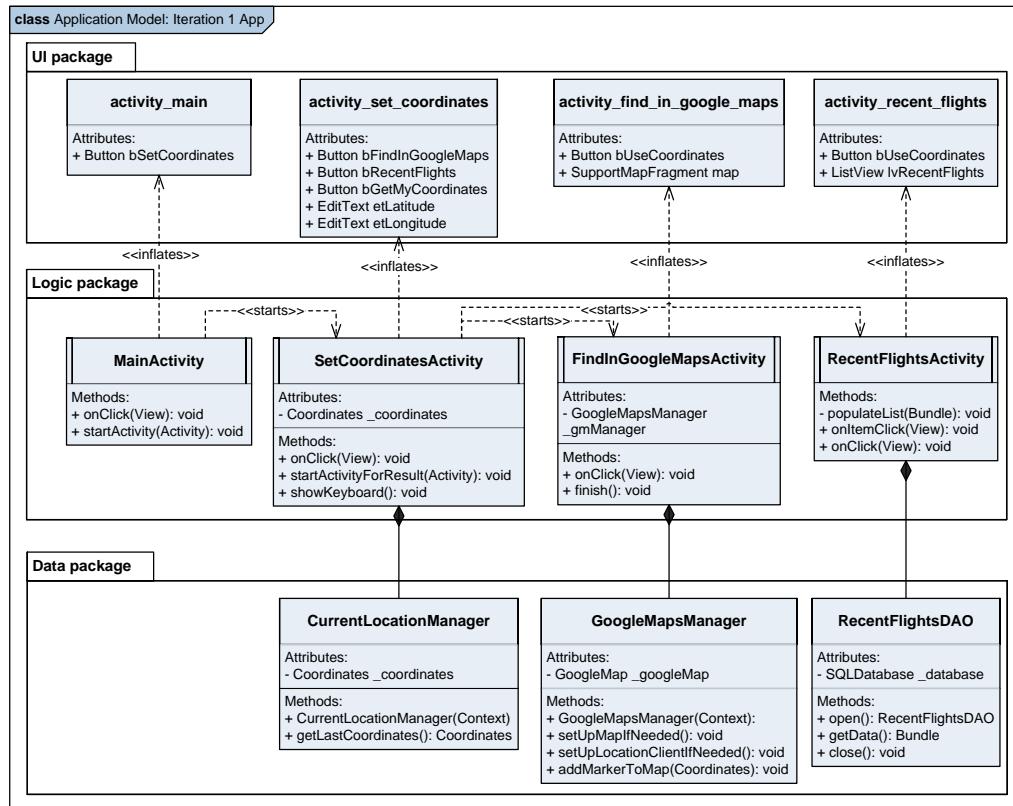
Klassediagram for app: Iteration 1

På figur 3.17 ses de softwareklasser, der er identificeret ud fra applikationsklasserne i applikationsmodellen for app'en.



Figur 3.17. Softwareklasser for app: Iteration 1.

På figur 3.18 ses klassediagrammet for app'en efter iteration 1.



Figur 3.18. Klassediagram for app: Iteration 1.

Tilføjede klasser i denne iteration:

Først beskrives det logiske lag, hvorefter UI laget og datalaget beskrives.

Logic

Alle klasser i det logiske lag er aktiviteter, der varetager ansvarsområder tilhørende controller-klassen *ChooseCoordinates*.

- **MainActivity:**

MainActivity varetager visning af hovedmenuen for brugeren, samt initialisering af underaktiviteter, når brugeren trykker på GUI'ets knapper.

- **SetCoordinatesActivity:**

Denne aktivitet står for at præsentere brugeren for de forskellige muligheder, der er for at vælge koordinater, og samtidig give brugeren mulighed for at overføre koordinaterne til serveren.

- **FindInGoogleMapsActivity:**

Denne aktivitet står for, at præsentere brugeren for et Google Maps kort, hvor brugeren kan markere en lokation. Aktiviteten står derudover for at returnere den markerede lokations koordinater til *SetCoordinatesActivity*.

- **RecentFlightsActivity:**

Denne aktivitet står for at populere en liste på GUI'et med data fra *RecentFlightsDAO*. Aktiviteten lader samtidig brugeren vælge et element på listen, og returnerer informationen fra denne liste til *SetCoordinatesActivity*.

UI

- **activity_main:**

Denne UI-klasse indeholder alle UI-elementer for *MainActivity*, dvs. alle elementer som brugeren bliver præsenteret for, heriblandt knapper, tekst osv.

- **activity_set_coordinates:**

Denne klasse indholder alle UI-elementer for *SetCoordinatesActivity*.

- **activity_find_in_google_maps:**

Denne klasse indeholder alle UI-elementer for *FindInGoogleMapsActivity*.

- **activity_recent_flights:**

Denne klasse indeholder alle UI-elementer for *RecentFlightsActivity*.

Data

- **CurrentLocationManager:**

Denne klasse håndterer GPS kommunikation, og står for at returnere brugerens nuværende koordinater til *SetCoordinatesActivity*.

- **GoogleMapsManager:**

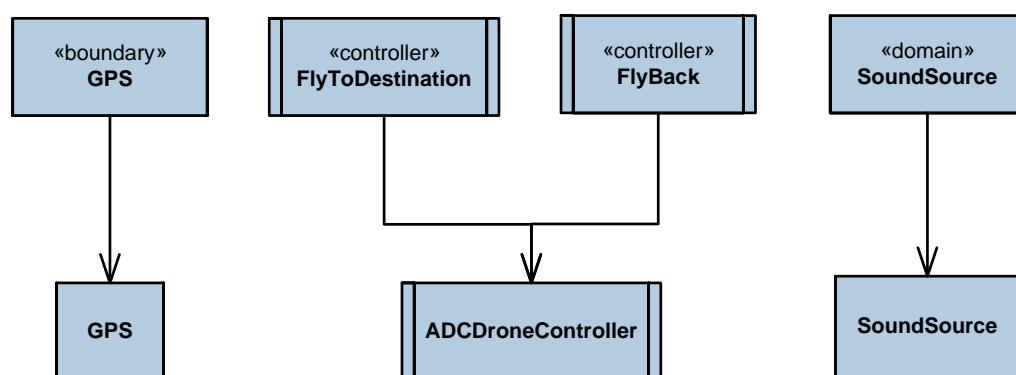
Denne klasse håndterer kommunikation med Google Maps' servere, og står for at hente kortet som vises i *FindInGoogleMapsActivity*.

- **RecentFlightsDAO:**

Denne klasse håndterer kommunikation med databasen, hvori tidligere aktiverede koordinater gemmes. Med "aktiverede" menes der afsendte koordinater til serveren.

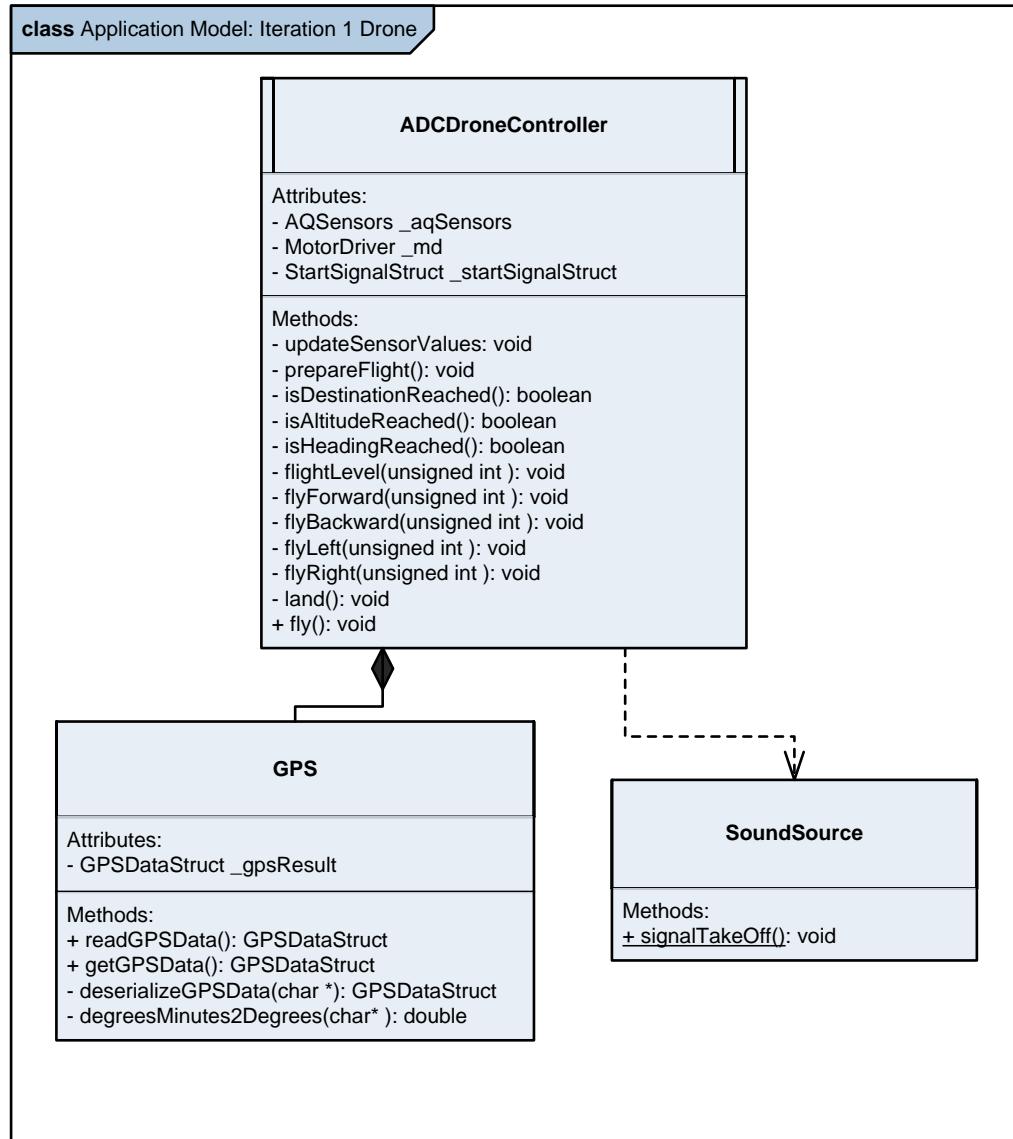
Klassediagram for drone: Iteration 1

På figur 3.19 ses de softwareklasser, der er identificeret ud fra applikationsklasserne i applikationsmodellen for dronen.



Figur 3.19. Softwareklasser for drone: Iteration 1.

På figur 3.20 ses klassediagrammet for dronen efter iteration 1.



Figur 3.20. Klassediagram for drone: Iteration 1.

Tilføjede klasser i denne iteration:

- **ADCDroneController:**

ADCDroneController er hovedklassen på dronen. Den udgøres af funktionaliteten fundet for klasserne *FlyToDestination* og *FlyBack*. Klassen indeholder de metoder der kaldes fra dronens main funktion.

- **GPS:**

GPS-klassen indeholder de metoder der skal anvendes for at modtage aktuel GPS information og returnere disse i en *GPSDataStruct*.

- **SoundSource:**

SoundSource-klassen anvendes til at signalere take-off.

3.3.2 Iteration 2

I iteration 2 tilføjes use case 11, 12 og 13. Heri ligger den basale kommunikation med serveren. Use case 11 og 12 beskriver serverens funktionalitet. Logical view'et for use case 11 og 12 beskrives derfor ud fra serverens perspektiv, jf. figur 3.2, hvor det, for use case 13, beskrives fra klientens perspektiv.

3.3.2.1 Use case 12: Modtag data fra klient

Domænemodellen er analyseret jf. use case 12 "Modtag data fra klient". På figur 3.21 ses de fundne klasser.



Figur 3.21. Fundne klasser for use case 12.

Klassebeskrivelser

Alle klasser er navngivet med præfixet "ADC", der står for "Autonomous Drone Control".

ADCClientIF: Denne klasse er serverens interface ud til den forbindende klient, og dermed den resterende del af ADC systemet.

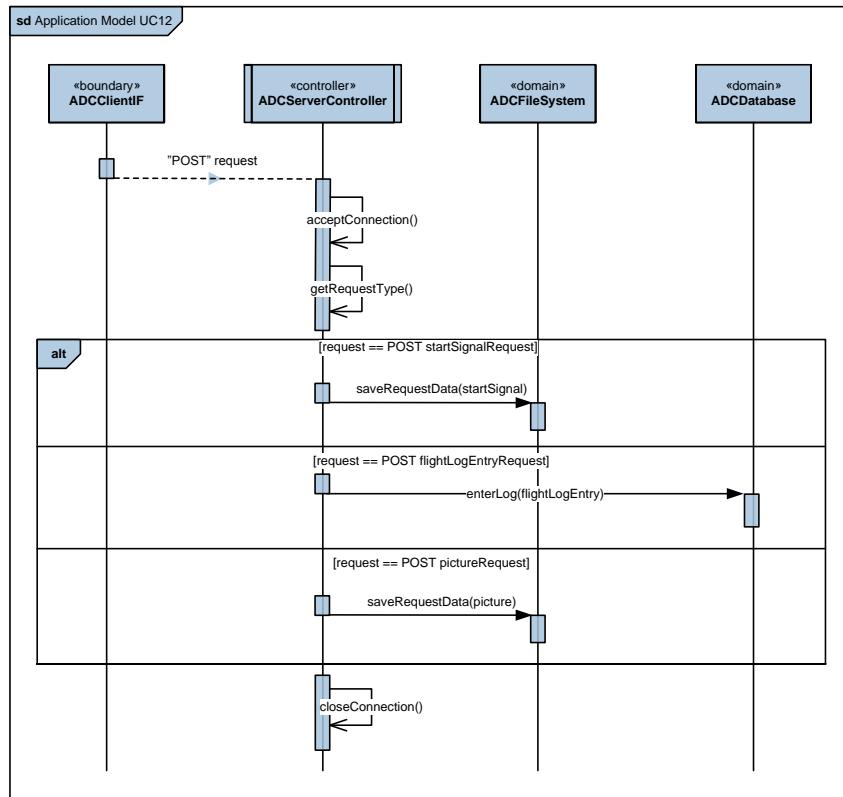
ADCServerController: Dette er controller-klassen for use-casen, der står for at koordinere al funktionalitet og kommunikation på serveren.

ADCFileSystem: Denne klasse står for al fil-IO på serveren, når der skal gemmes eller hentes filer som f.eks. billeder.

ADCDatabase: Denne klasse står for al interaktion med databasen på serveren.

Sekvensdiagram

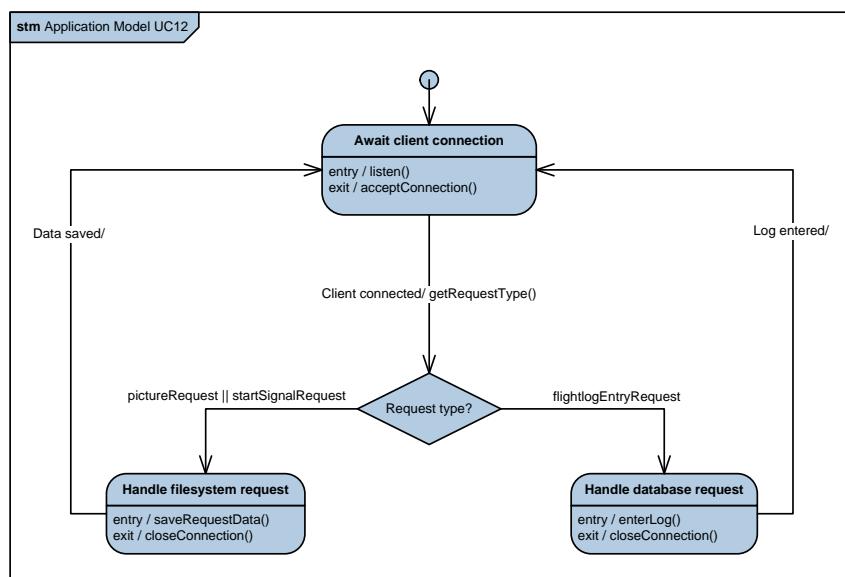
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen. Derudover identifierer sekvensdiagrammet nogle af de essentielle metoder og attributter for klasserne.



Figur 3.22. Sekvensdiagram for use case 12.

Tilstandsdiagram

På figur 3.23 ses tilstandsdiagrammet for controller-klassen. Diagrammet beskriver hvordan serveren håndterer forespørgsler på forskellige typer af data.



Figur 3.23. Tilstandsdiagram for ADCServerController.

3.3.2.2 Use case 11: Send data til klient

Domænemodellen er analyseret jf. use case 11 "Send data til klient". På figur 3.24 ses de fundne klasser.



Figur 3.24. Fundne klasser for use case 11.

De fundne klasser for use case 11 er de samme som de fundne klasser for use case 12. Den eneste forskel på de to use cases er, hvorvidt klienten forespørger data på serveren, eller forespørger at sende data til serveren, og derfor vil de overordnede klasser være de samme. Forskellen ligger i håndteringen af klientens forespørgsel.

Klassebeskrivelser

Alle klasser er navngivet med et prefix, "ADC", der står for "Autonomous Drone Control".

ADCClientIF: Denne klasse er serverens interface ud til den forbindende klient, og dermed den resterende del af ADC systemet.

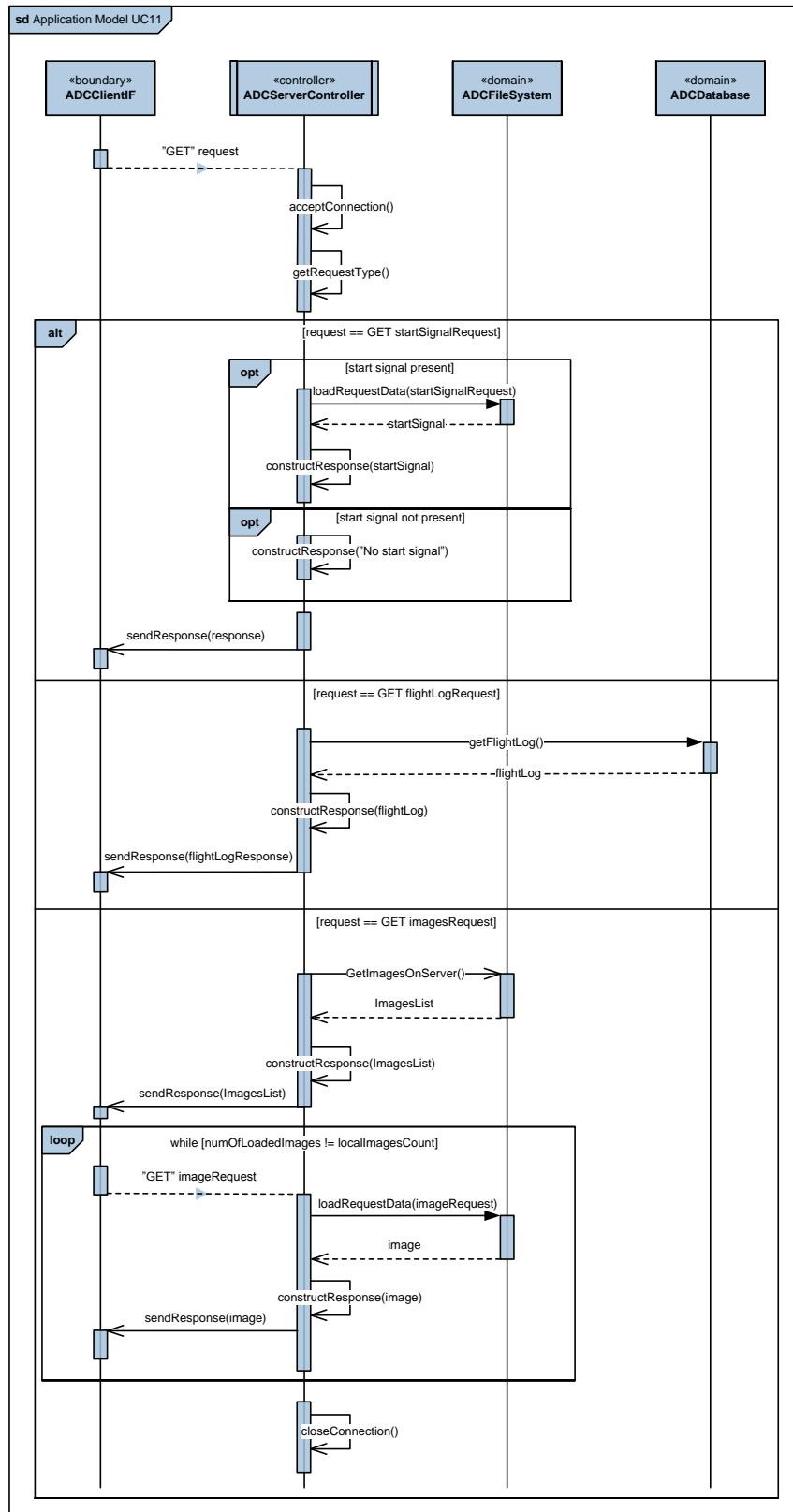
ADCServerController: Dette er controller-klassen for use-casen, der står for at koordinere al funktionalitet og kommunikation på serveren.

ADCFileSystem: Denne klasse står for al fil-IO på serveren, når der skal gemmes, eller hentes filer, som f.eks. billeder.

ADCDatabase: Denne klasse står for al interaktion med databasen på serveren.

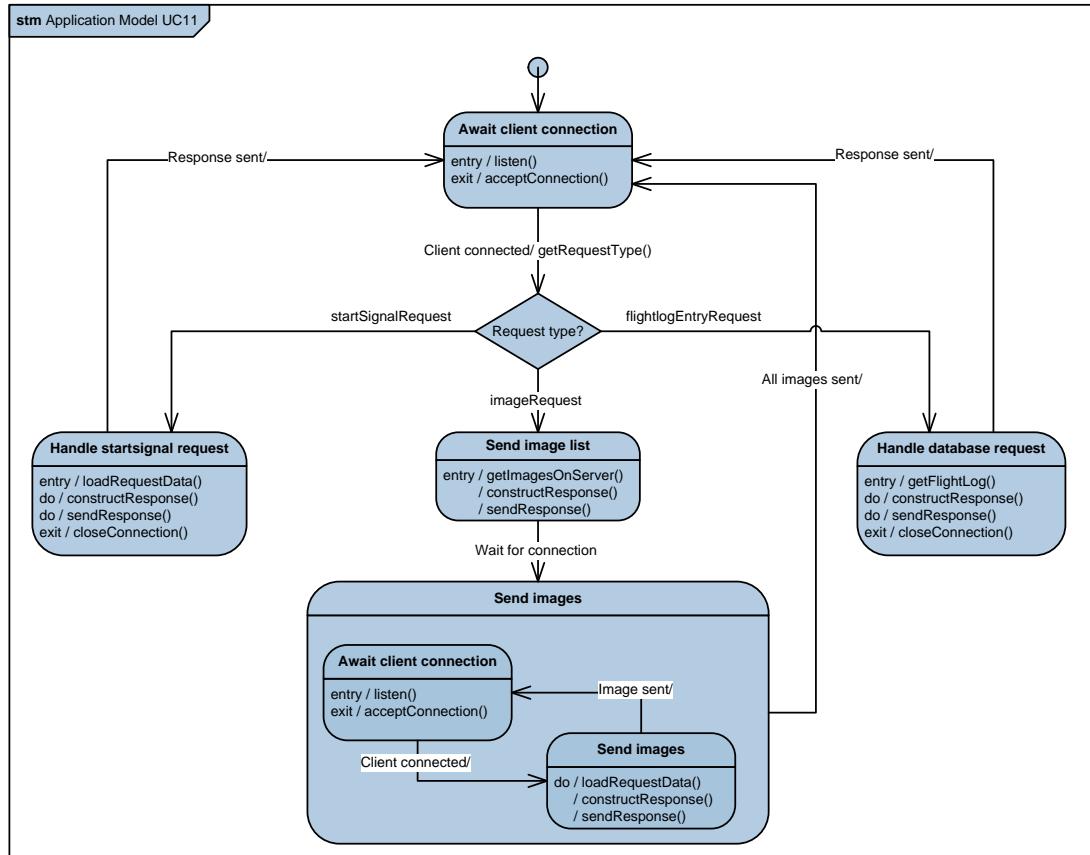
Sekvensdiagram

Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen. Derudover identificerer sekvensdiagrammet nogle af de essentielle metoder og attributter for klasserne.

**Figur 3.25.** Sekvensdiagram for use case 11.

Tilstandsdiagram

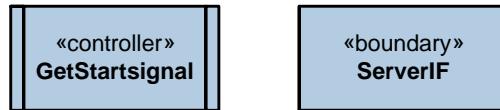
På figur 3.26 ses tilstandsdiagrammet for controller-klassen. Diagrammet beskriver, hvordan serveren håndterer forespørgsler på forskellige typer af data, som klienten vil sende til serveren.



Figur 3.26. Tilstandsdiagram for ADCServerController.

3.3.2.3 Use case 13: Hent startsignal

Domænemodellen er analyseret jf. use case 13 "Hent startsignal". På figur 3.27 ses de fundne klasser.



Figur 3.27. Fundne klasser for use case 13.

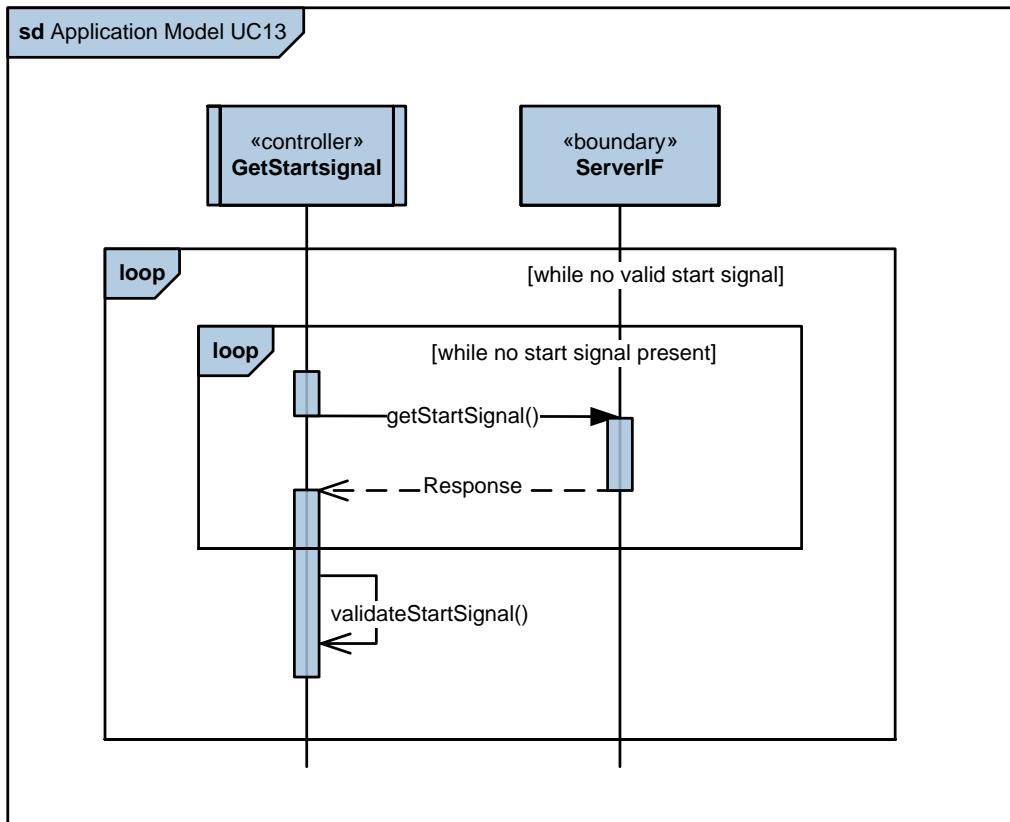
Klassebeskrivelser

GetStartsignal: *GetStartsignal* er en controller-klasse, der står for at koordinere al funktionalitet i use casen.

ServerIF: *ServerIF* er en boundary-klasse. Den står for al kommunikation til ADC-server.

Sekvensdiagram

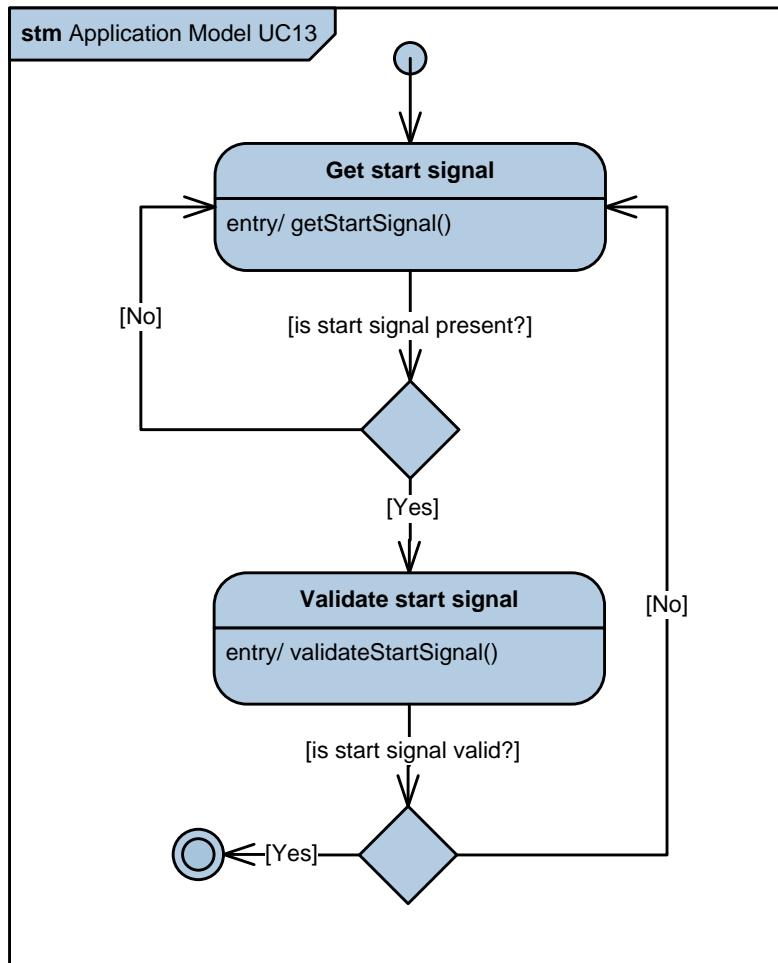
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen.



Figur 3.28. Sekvensdiagram for use case 13.

Tilstandsdiagram

På figur 3.29 ses tilstandsdiagrammet for use case 13. Diagrammet beskriver hentning og validering af startsignal.



Figur 3.29. Tilstandsdiagram for use case 13.

3.3.2.4 Klassediagrammer

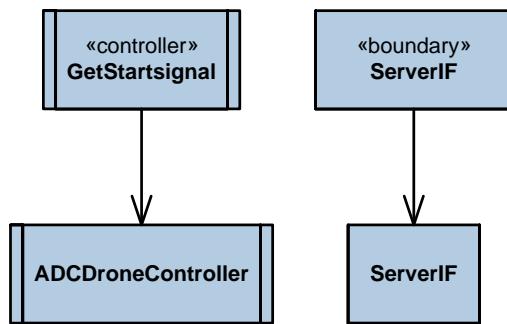
Ved analyse af af applikationsmodellerne for use case 11, 12 og 13, resulterer iteration 2 i følgende klassediagrammer. Den implementerede software til iterationen, er bygget op omkring disse klassediagrammer, og eventuelle afvigelser fra den implementerede kode er kun opdateret i klassediagrammet og altså ikke i sekvens- og tilstandsdiagrammerne. De klasser, der blev identificeret i iteration 1, er blevet opdateret, hvis der er fundet nye metoder eller attributter. De opdaterede klasser, og eventuelle nye klasser der er kommet til i iterationen, er fremhævet.

Klassediagram for app: Iteration 2

Der er ikke fundet nye metoder eller attributter for app'en i denne iteration. Se derfor figur 3.18

Klassediagram for drone: Iteration 2

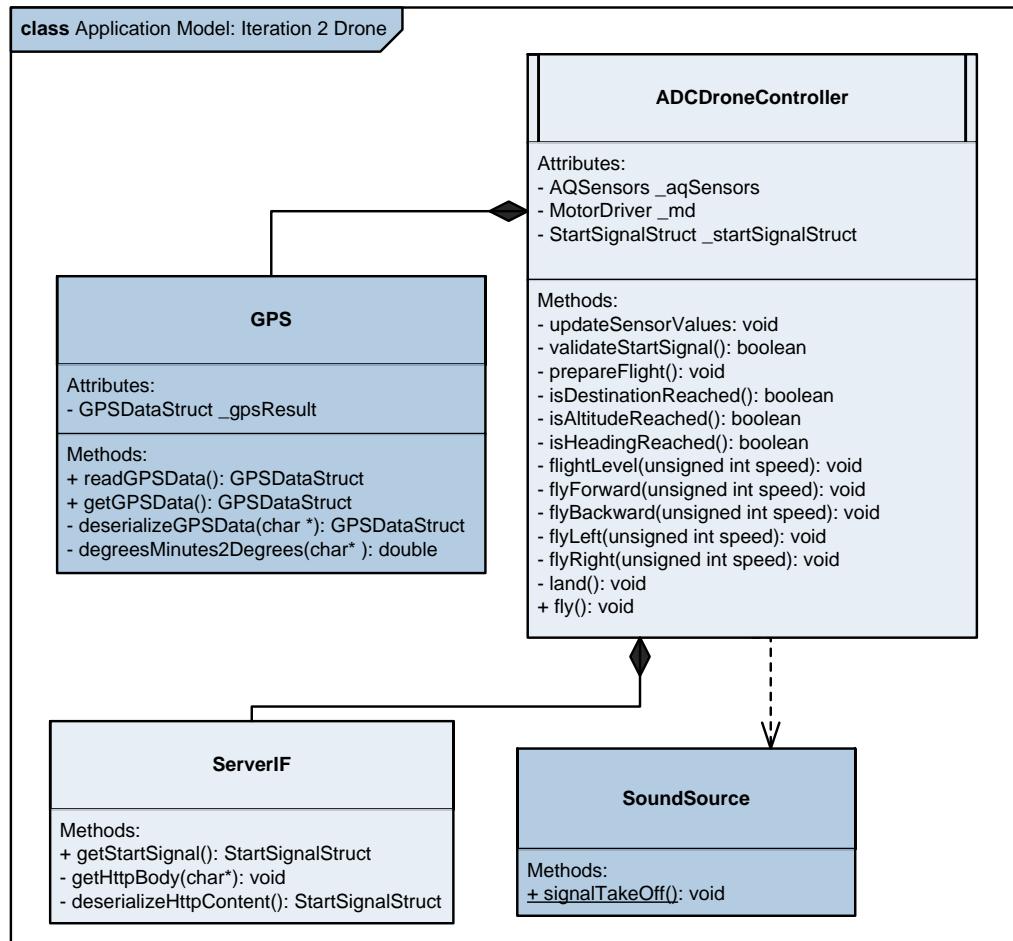
På figur 3.30 ses de softwareklasser, der er identificeret ud fra applikationsklasserne i applikationsmodellen for dronen.



Figur 3.30. Softwareklasser for drone: Iteration 2.

Som det ses på figuren, er der tilføjet en softwareklasse *ServerIF*. Funktionaliteten i controller-klassen *GetStartsignal*, der validerer startsignalet, er tilføjet til den allerede eksisterende softwareklasse *ADCDroneController*.

På figur 3.31 ses klassediagrammet for dronen efter iteration 2.



Figur 3.31. Klassediagram for drone: Iteration 2.

Tilføjede klasser i denne iteration:

- **ServerIF:**

Denne klasse varetager kommunikationen med serveren, og benyttes i denne iteration til at modtage et startsignal fra serveren.

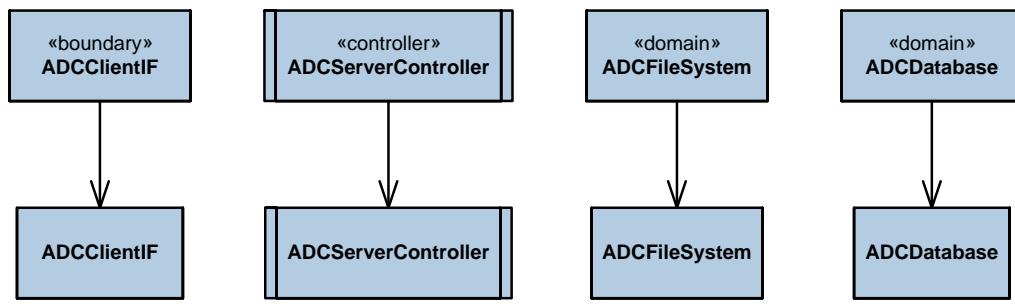
Opdaterede klasser i denne iteration:

- **ADCDroneController:**

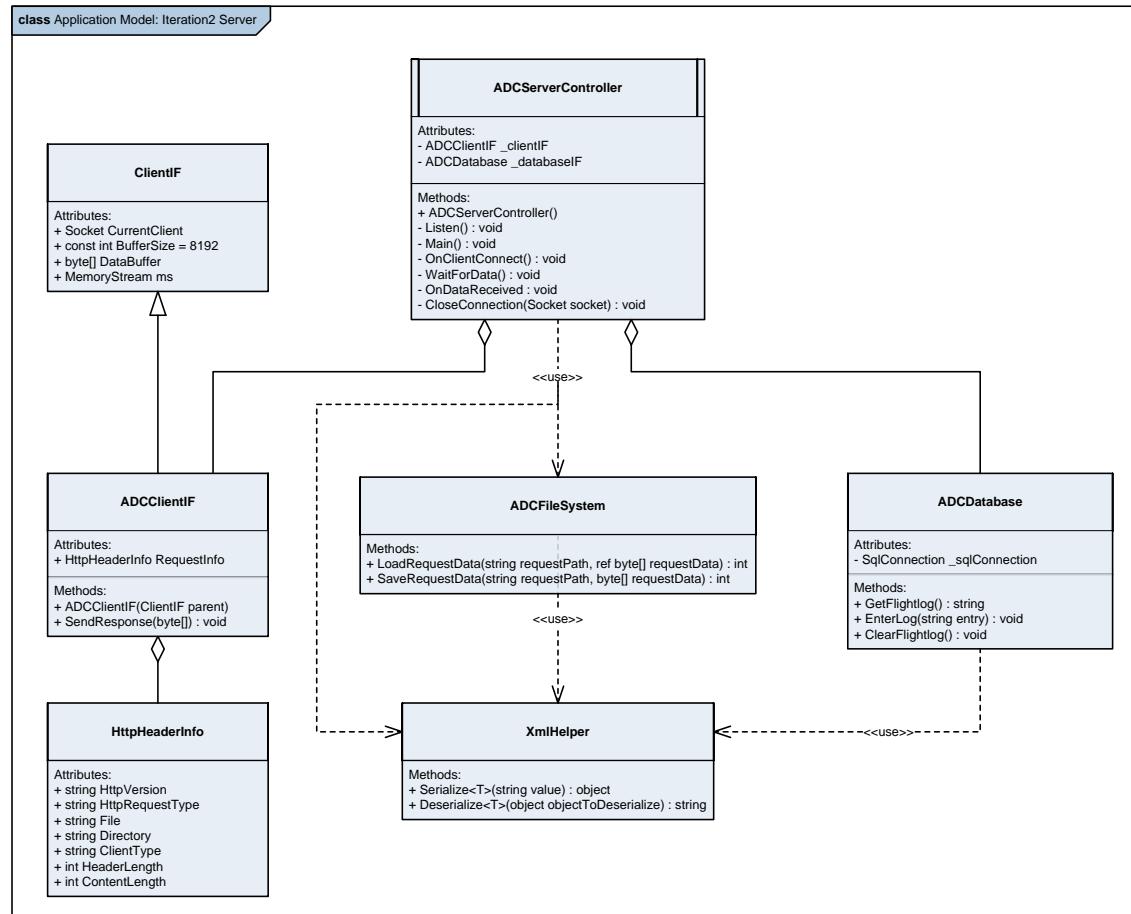
Denne klasse er opdateret med funktionen *validateStartsignal*, der benyttes, når et startsignal er modtaget fra serveren, og det skal valideres.

Klassediagram for server: Iteration 2

På figur 3.32 ses de softwareklasser, der er identificeret ud fra applikationsklasserne i applikationsmodellen for serveren.

**Figur 3.32.** Softwareklasser for server: Iteration 2.

På figur 3.33 ses iterationens endelige klassediagram for serveren. Da serveren er skrevet i C#, er alle datatyper C#-specifikke. Udenfor identificerede klasser fra applikationsmodellen er der tilføjet hjælpeklasser til klassediagrammet, for at isolere funktionalitet der benyttes flere steder i systemet, og for at uddeleger ansvar de øvrige klasser ellers skulle have haft.

**Figur 3.33.** Klassediagram for server: Iteration 2.

Tilføjede klasser i denne iteration:

- **ADCServerController:**

Denne klasse er main-klassen på serveren (dvs. den indeholder main-funktionen, og

eksekveres som det første i programmet), og er en direkte oversættelse af controller-klassen for use case 11 og 12. Den står for at initialisere server-socket'en og starte lyttetråden. Den indeholder funktioner til håndtering af forbindende klienter og deres forespørgsler, og står generelt for at koordinere kommunikationen mellem de øvrige klasser på serveren. Klassen benytter hjælpeklassen *XmlHelper* til at verificere et modtaget startsignal, ved at serialisere - og senere deserialisere det⁵.

- **ADCFileSystem:**

Denne klasse håndterer al fil-I/O på serveren, og benyttes hver gang en fil skal gemmes, elle hentes, på serverens harddisk. Den står samtidig for at mappe virtuelle mappestier i HTTP-headere fra klienten, til lokale mappestier på serveren, hvor filer bliver gemt/hentet. Klassen benytter hjælpeklassen *XmlHelper* til at deserialisere startsignalet før det skal gemmes i serverens filesystem.

- **ADCDatabase:**

Denne klasse står for forbindelsen til SQL-serveren, og indeholder funktioner til at tilgå databasen. Klassen benytter hjælpeklassen *XmlHelper* til at serialisere logposter, og deserialisere flight log'en.

- **ADCClientIF:**

Denne klasse nedarver fra *ClientIF*, og varetager selve forbindelsen til klienten. Klassen indeholder al information om klienten, samt metoder til at sende svar tilbage til denne, efter forespørgslen er blevet håndteret.

De tilføjede hjælpeklasser er:

- **ClientIF:**

Denne klasse indeholder de essentielle komponenter, der er nødvendige for at kunne oprette forbindelse til, og modtage data fra, en forbindende klient. Klassen udgør den forbindende klients tilstand, og benyttes hver gang en klient forbinder til serveren. Den indeholder et socket-objekt, der udgør selve forbindelsen til klienten, og styrer al kommunikation med denne. Samtidig indeholder klassen en buffer til at indeholde data modtaget fra klienten. Slutelig benytter klassen et objekt af typen *MemoryStream* til at styre datastrømmen fra klienten og gemme denne i en *DataBuffer*.

- **HttpHeaderInfo:**

ADCClientIF-klassen benytter denne klasse til at indeholde de oplysninger, der forefindes i den modtagne HTTP-header fra klienten. Når en klient er forbundet, og data er modtaget fra den, deserialiseres HTTP-header'en, og oplysningerne gemmes i et *HttpHeaderInfo*-objekt.

- **XmlHelper:**

XmlHelper-klassen benyttes til at deserialisere XML-strenge til forskellige typer af objekter, og til at serialisere objekter til XML-strenge. *XmlHelper* indeholder statiske funktioner, der kan benyttes uden at oprette en instans af klassen. Klassen benyttes hver gang data skal gemmes i, eller hentes fra, databasen eller filesystemet.

⁵Se underafsnit 3.6.4.2 i afsnit 3.6.3 "Beskrivelse af dataformat" for yderligere informationer om denne proces.

3.3.3 Iteration 3

I iteration 3 tilføjes use case 2, 4 og 5. I denne iteration udvides kommunikationen med serveren. I alle use casenes tilfælde vil logical view'et blive beskrevet fra klientens perspektiv, som det ses på figur 3.2, da disse use cases beskriver app'ens og dronens måder at forbinde til serveren på.

3.3.3.1 Use case 2: Upload af startsignal

Use case 2 beskriver, hvordan de koordinater brugeren har valgt i app'en, bliver uploadet til serveren. Use casen beskriver scenariet set fra app'ens synspunkt, hvilket betyder, at selve serveren ikke er med i use casen - men interfacet til den er. Ud fra use casen og domænemodellen er klasserne på figur 3.34 identificeret.



Figur 3.34. Fundne klasser for use case 2.

Klassebeskrivelser

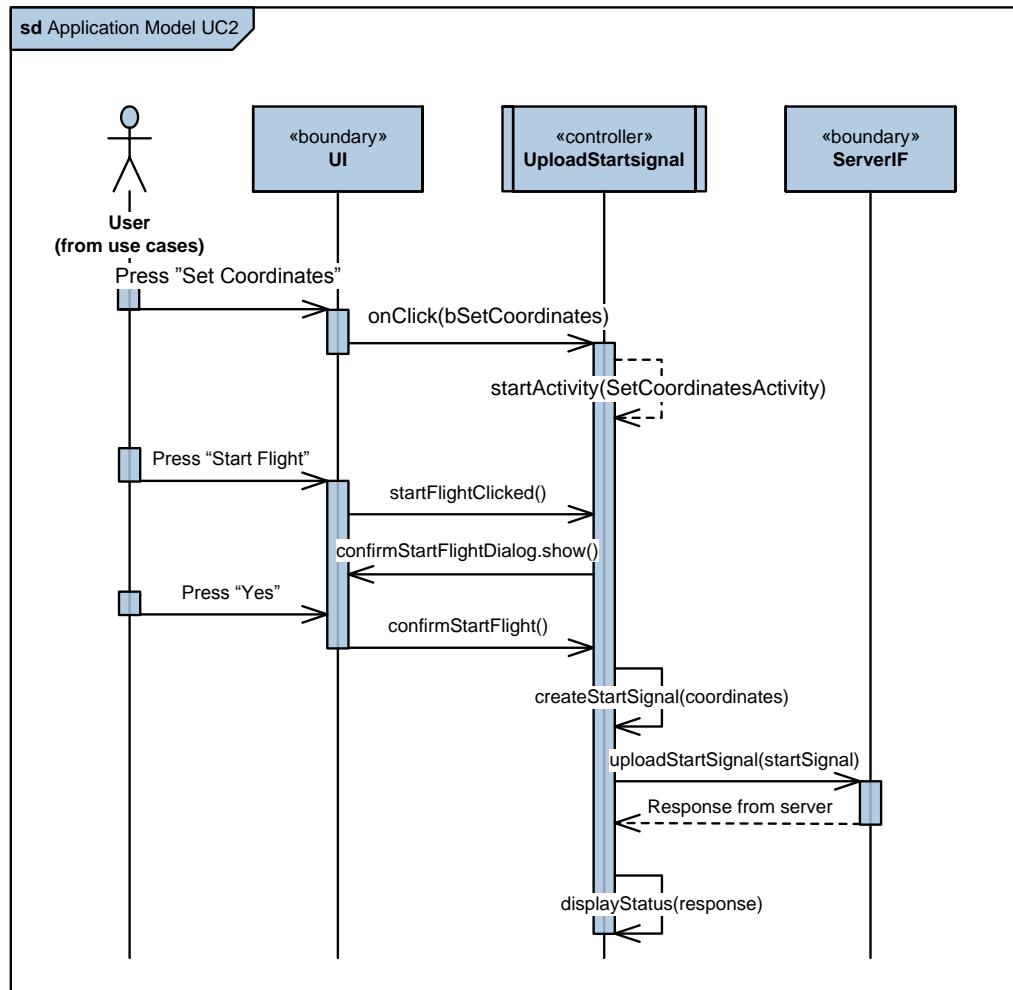
UI: *UI* er en boundary-klasse, der indeholder de grafiske elementer i app'en, og bestemmer således app'ens fremtoning over for brugeren. *UI*-klassen indeholder ingen funktionalitet. I dette system er det meste af *UI*'et implementeret statisk, der for Java Android betyder, at det er implementeret som xml-filer. *UI*'et har et stort ansvarsområde, og implementeres derfor som flere xml-filer. *UI* er således ikke en klasse, men en pakke bestående af flere grafiske klasser. Klasserne i pakken vil senere blive udspecifieret.

UploadStartsignal: *UploadStartsignal* er controller-klassen i use casen, og står for at koordinere al kommunikation og funktionalitet i use casen. Klassen vil i dette afsnit blive beskrevet som en selvstændig klasse, mens funktionaliteten vil blive tilføjet til det opdaterede app-klassediagram for iteration 3 under klassen *SetCoordinatesActivity*, da denne klasse også varetager *UploadStartsignal*'s ansvarsområde.

ServerIF: Denne klasse er interfacet til serveren, og administrerer al kommunikation med denne.

Sekvensdiagram

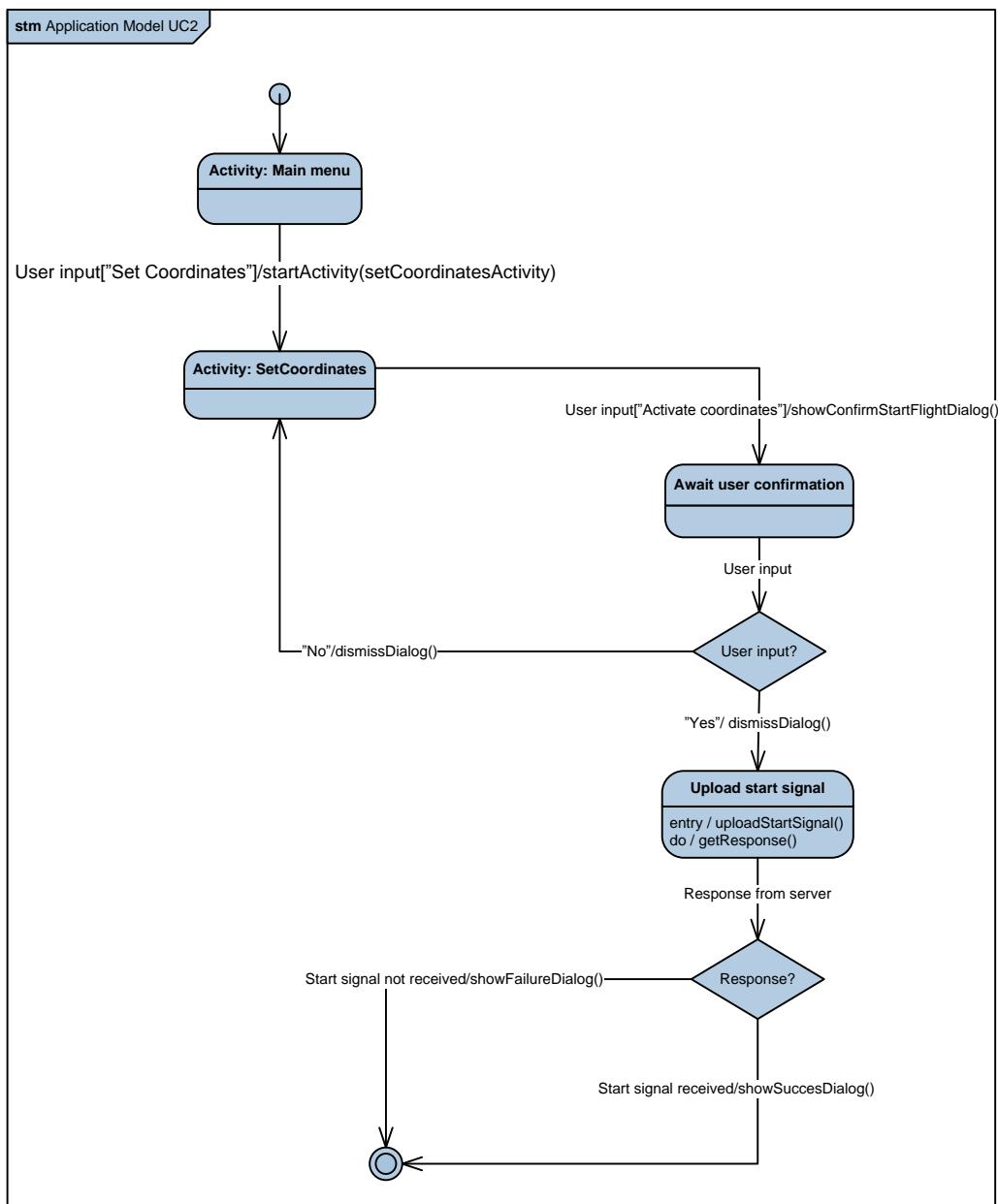
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen. Derudover identifierer sekvensdiagrammet nogle af de essentielle metoder og attributter for klasserne. På figur 3.35 ses det overordnede sekvensdiagram for use casen.



Figur 3.35. Sekvensdiagram for use case 2.

Tilstandsdiagram

På figur 3.36 ses tilstandsdiagrammet for use case 2. Diagrammet består af forskellige tilstande, hvor nogle er noteret med "Activity" for at vise, at konteksten i disse tilfælde er en Android aktivitet.



Figur 3.36. Tilstandsmaskine for use case 2.

3.3.3.2 Use case 5: Logning af flyvedata fra drone

Domænemodellen er analyseret jf. use case 5 "Logning af flyvedata fra drone". På figur 3.37 ses de fundne klasser.



Figur 3.37. Fundne klasser for use case 5.

Klassebeskrivelser

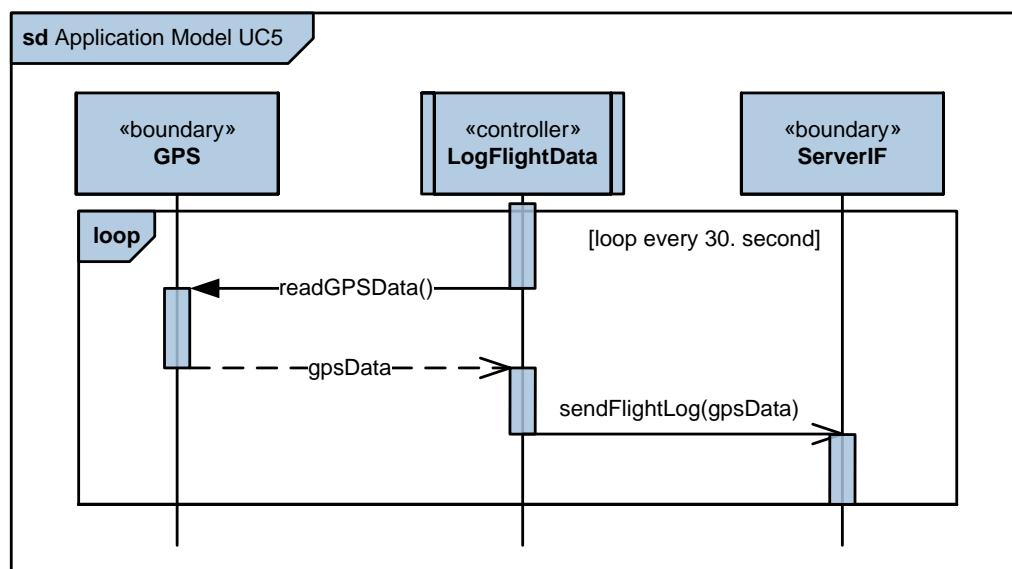
LogFlightData: *LogFlightData* er en controller-klasse, der står for at koordinere al funktionalitet i use casen.

GPS: *GPS*-klassen er en boundary-klasse, der står for Arduino'ens kommunikation med GPS-satellitterne. Den indeholder metoder til at modtage lokationsdata.

ServerIF: Denne klasse er interfacet til serveren, og administrerer al kommunikation med denne.

Sekvensdiagram

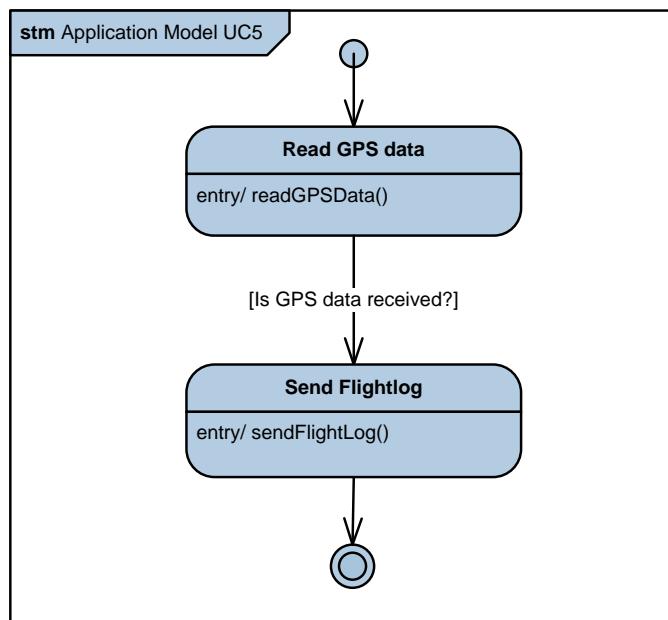
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen.



Figur 3.38. Sekvensdiagram for use case 5.

Tilstandsdiagram

På figur 3.39 ses tilstandsdiagrammet for use case 5. Diagrammet beskriver aflæsning af GPS-information samt afsendelse af logposter. Dette gentages med et 30. sekunders interval.



Figur 3.39. Tilstandsdiagram for use case 5.

3.3.3.3 Use case 4: Visning af flightlog

Use case 4 beskriver, hvordan brugeren vælger at se flight log'en i app'en. Ud fra use casen og domænemodellen er klasserne på figur 3.40 identificeret.



Figur 3.40. Fundne klasser for use case 4.

Klassebeskrivelser

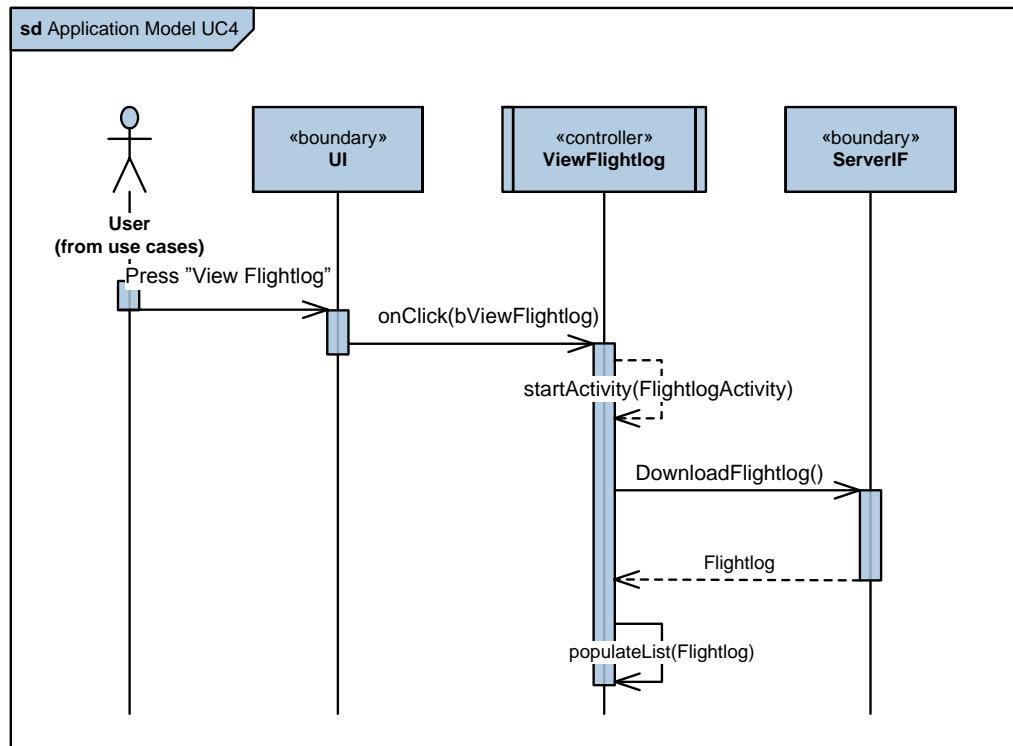
UI: *UI* er en boundary-klasse, der indeholder de grafiske elementer i app'en, og bestemmer således app'ens fremtoning over for brugeren. *UI*-klassen indeholder ingen funktionalitet. I dette system er det meste af *UI*'et implementeret statisk, der for Java Android betyder, at det er implementeret som xml-filer. *UI*'et har et stort ansvarsområde, og implementeres derfor som flere xml-filer. *UI* er således ikke en klasse, men en pakke bestående af flere grafiske klasser. Klasserne i pakken vil senere blive udspesificeret.

ViewFlightlog: *ViewFlightlog* er en controller-klasse, der står for at koordinere al kommunikation og funktionalitet i use case'en. *ViewFlightlog* varetager både logiske og UI-relatedede operationer, og inddeltes derfor senere i flere klasser for at uddeletere dens ansvarsområde. Den beskrives dog i dette afsnit som en selvstændig klasse.

ServerIF: Denne klasse er interfacet til serveren, og administrerer al kommunikation med denne.

Sekvensdiagram

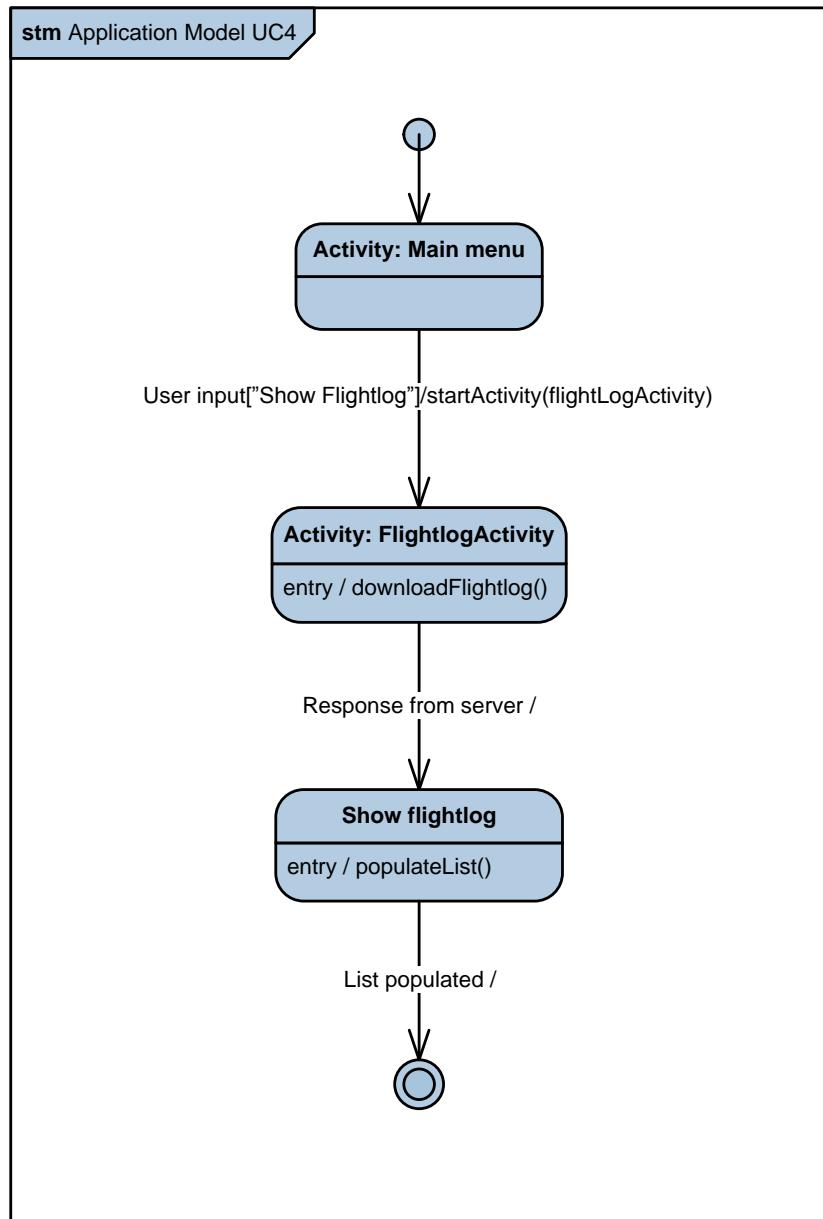
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen. Derudover identificerer sekvensdiagrammet nogle af de essentielle metoder og attributter for klasserne. På figur 3.41 ses det overordnede sekvensdiagram for use casen.



Figur 3.41. Sekvensdiagram for use case 4.

Tilstandsdiagram

På figur 3.42 ses tilstandsdiagrammet for use case 4. Diagrammet består af forskellige tilstande, hvor nogle er noteret med "Activity" for at vise, at konteksten i disse tilfælde er en Android aktivitet.



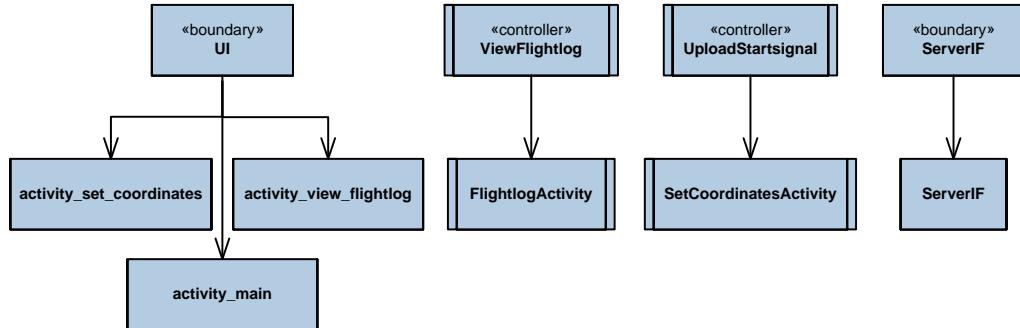
Figur 3.42. Tilstandsmaskine for use case 4.

3.3.3.4 Klassediagrammer

Ved analyse af af applikationsmodellerne for use case 2, 4 og 5 resulterer iteration 3 i følgende klassediagrammer. Den implementerede software til iterationen er bygget op omkring disse klassediagrammer, og eventuelle afvigelser fra den implementerede kode er kun opdateret i klassediagrammet og altså ikke i sekvens- og tilstandsdiagrammerne. De klasser der blev identificeret i iteration 1 og 2, er blevet opdateret, hvis der er fundet nye metoder eller attributter. De opdaterede klasser, og eventuelle nye klasser der er kommet til i iterationen, er fremhævet.

Klassediagram for app: Iteration 3

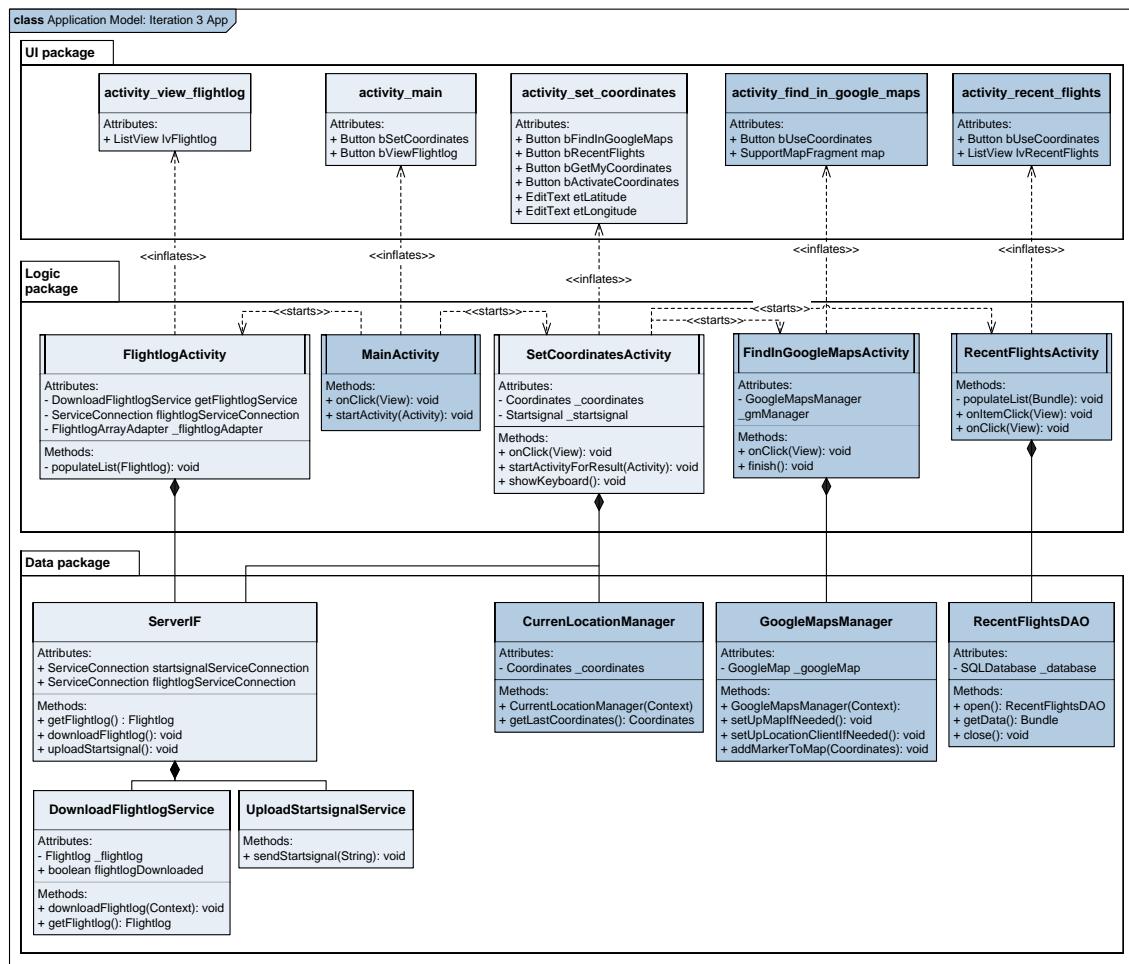
På figur 3.43 ses de softwareklasser, der er identificeret ud fra applikationsklasserne i denne iterations applikationsmodeller for app'en.



Figur 3.43. Softwareklasser for app: Iteration 3.

Der er tilføjet funktionalitet til den allerede eksisterende softwareklasse *ServerIF*. Funktionaliteten i controller-klassen *UploadStartsignal* er tilføjet til den allerede eksisterende softwareklasse *SetCoordinatesActivity*, mens der er tilføjet UI-elementer til de to eksisterende UI-klasser *activity_main* og *activity_set_coordinates*.

På figur 3.44 ses klassediagrammet for app'en i iteration 3. Udeover identificerede klasser fra applikationsmodellen er der tilføjet hjælpeklasser til klassediagrammet for at isolere funktionalitet, der benyttes flere steder i systemet, og for at uddeletere ansvar som de øvrige klasser ellers skulle have haft.



Figur 3.44. Klassediagram for app: Iteration 3.

Tilføjede klasser i denne iteration:

- **FlightlogActivity:**

Denne klasse varetager de logiske og UI-relaterede operationer, for den funktionalitet der er tilføjet i denne iteration.

- **activity_view_flightlog:**

Denne klasse indeholder de UI-elementer brugeren præsenteres for.

- **ServerIF:**

Denne klasse varetager kommunikation med serveren. `ServerIF` benytter sig af de to hjælpeklasser `UploadStartsignalService` og `DownloadFlightlogService` til henholdsvis at uploadede startsignal og downloadede flight log.

Opdaterede klasser i denne iteration:

- **SetCoordinatesActivity:**

Opdateret til at kunne konstruere et startsignal, og har fået tilføjet en instans af `ServerIF`-objektet, så den kan kommunikere med serveren.

- **activity_main:**

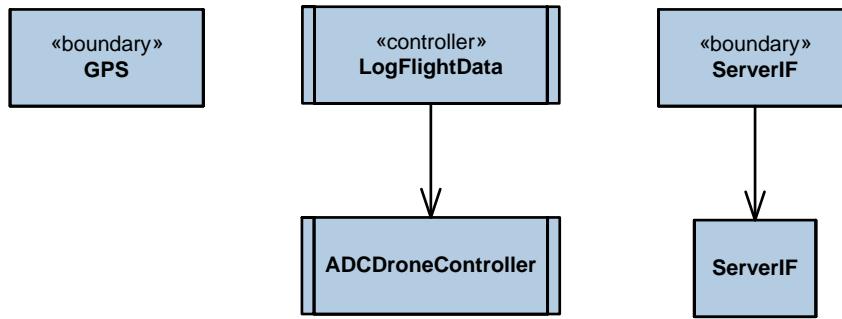
Opdateret med knap til at åbne `FlightlogActivity`.

- **activity_set_coordinates:**

Opdateret med knap til at aktivere koordinater.

Klassediagram for drone: Iteration 3

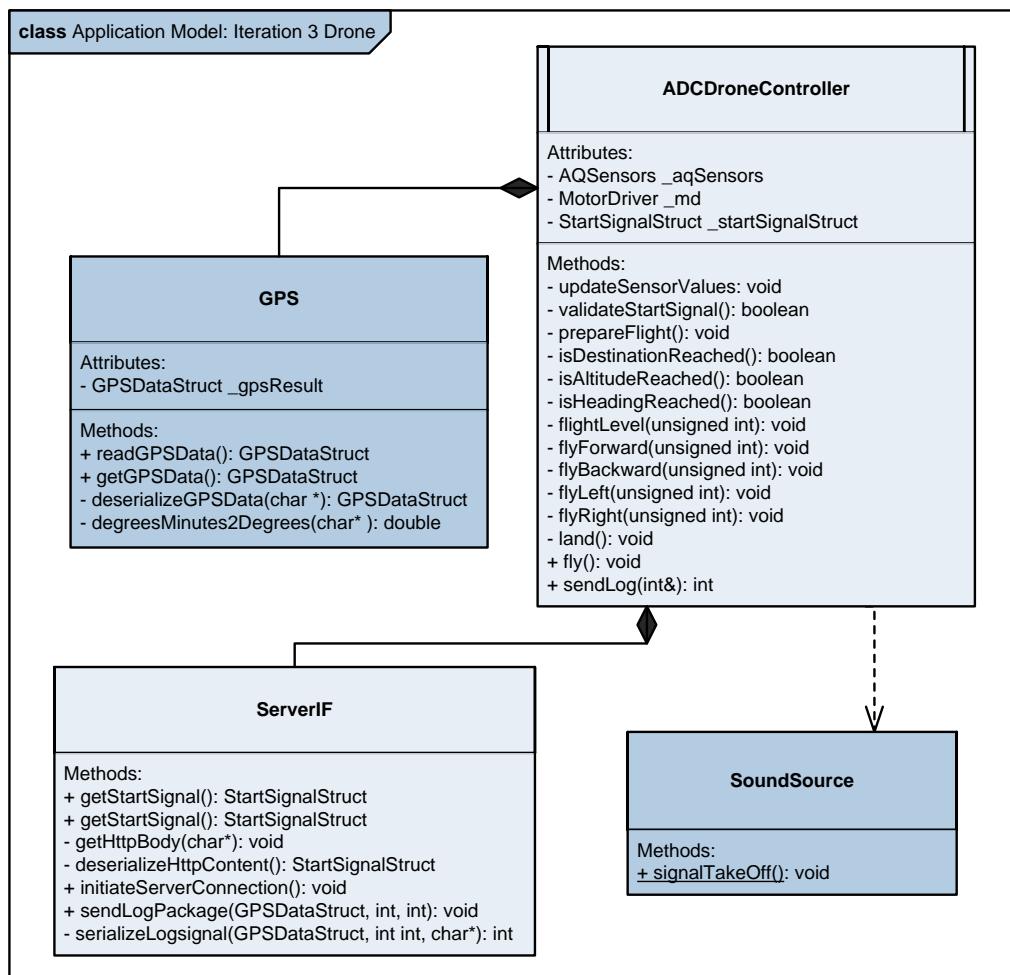
På figur 3.45 ses de softwareklasser, der er identificeret ud fra applikationsklasserne i applikationsmodellen for dronen.



Figur 3.45. Softwareklasser for drone: Iteration 3.

I denne iteration er der ikke tilføjet nye klasser. Der er derimod tilføjet funktionalitet til de to eksisterende klasser *ADCDroneController* og *ServerIF*. *GPS*-klassen benyttes i applikationsmodellen for use case 5.

På figur 3.46 ses klassediagrammet for dronen i iteration 3.



Figur 3.46. Klassediagram for drone: Iteration 3.

Opdaterede klasser i denne iteration:

- **ADCDroneController:**

Opdateret til at indeholde en funktion, `sendLog()`, der står for at sende logs til serveren ved bl.a. at kalde `ServerIF`'s `sendLogPackage()`.

- **ServerIF:**

Opdateret til at indeholde funktionerne, `initiateServerConnection()` og `sendLogPackage()`, der står for at sende logs til serveren.

Klassediagram for server: Iteration 3

Der er ikke fundet nye klasser, metoder eller attributter for serveren i denne iteration. For detaljeret klassediagram for serveren, se derfor figur 3.33.

3.3.4 Iteration 4

Denne iteration forventes ikke implementeret, hvorfor der ikke foreligger nogen dokumentation for alle iterationens use cases. Der er dog lavet et udkast til use case 3: "Vis billeder", der muliggør visning af billeder fra serveren i app'en, selvom use case 6: "Fotografering og upload af billede til server" ikke er implementeret. Use case 3 udvider app'en og kommunikationen mellem den og serveren, og beskrives fra klientens perspektiv, som det ses på figur 3.2.

3.3.4.1 Use case 3: Visning af billeder

Use case 3 beskriver, hvordan brugeren vælger at se flight log'en i app'en. Ud fra use casen og domænemodellen er klasserne på figur 3.47 identificeret.



Figur 3.47. Fundne pakker for use case 3.

Klassebeskrivelser

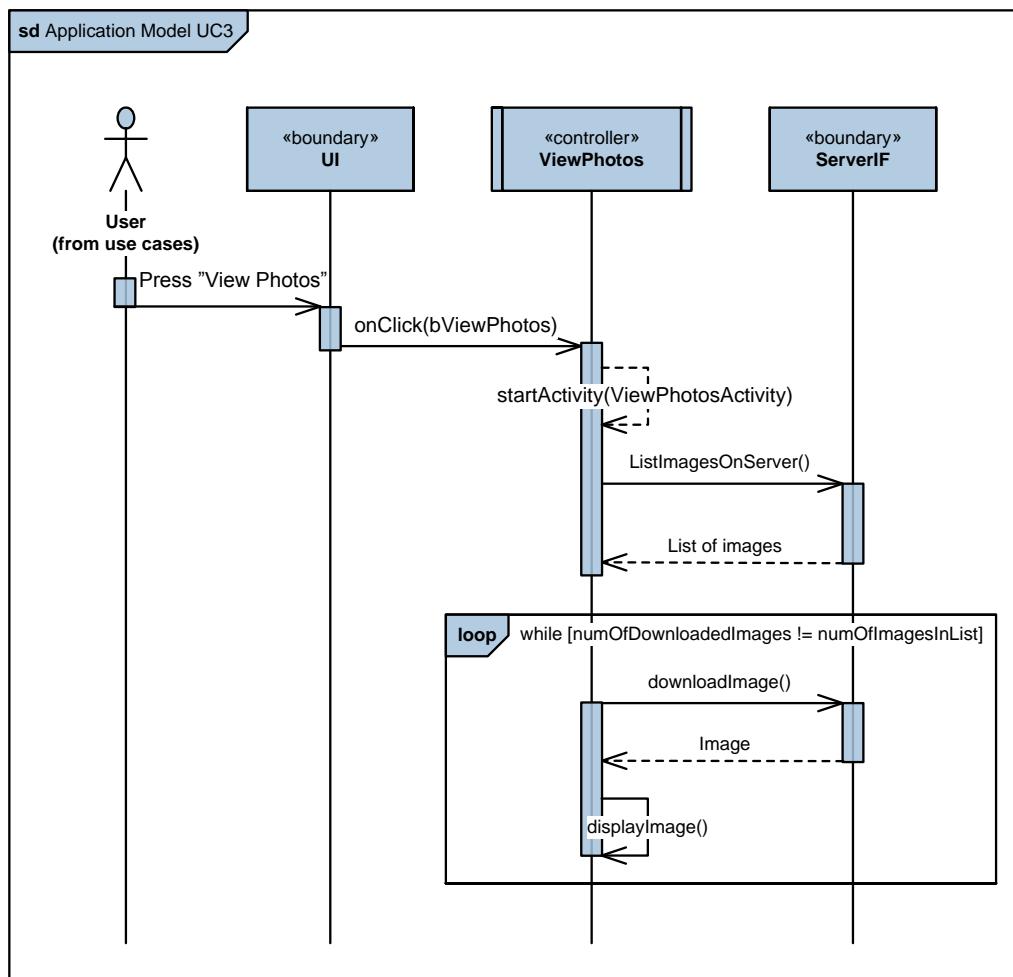
UI: *UI* er en boundary-klasse, der indeholder de grafiske elementer i app'en, og bestemmer således app'ens fremtoning over for brugeren. *UI*-klassen indeholder ingen funktionalitet. I dette system er det meste af *UI*'et implementeret statisk, der for Java Android betyder, at det er implementeret som xml-filer. *UI*'et har et stort ansvarsområde, og implementeres derfor som flere xml-filer. *UI* er således ikke en klasse, men en pakke bestående af flere grafiske klasser. Klasserne i pakken vil senere blive udspecifieret.

ViewPhotos: *ViewPhotos* er en controller-klasse, der står for at koordinere al kommunikation og funktionalitet i use casen. *ViewPhotos* står for at igangsætte download af billeder, samt at opdatere *UI*'et med billederne.

ServerIF: Denne klasse er interfacet til serveren, og administrerer al kommunikation med denne.

Sekvensdiagram

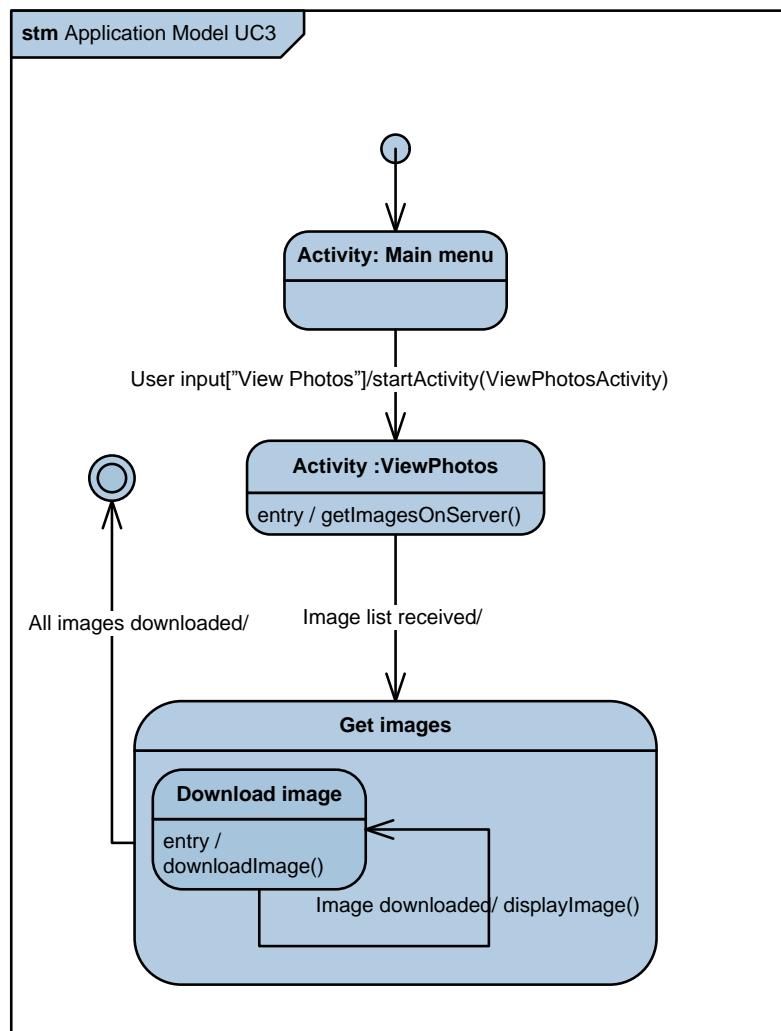
Ud fra de fundne overordnede klasser laves der et sekvensdiagram, der beskriver hvordan klasserne aktiveres under eksekvering af hovedforløbet i use casen. Derudover identificerer sekvensdiagrammet nogle af de essentielle metoder og attributter for klasserne. På figur 3.48 ses det overordnede sekvensdiagram for use casen.



Figur 3.48. Sekvensdiagram for use case 3.

Tilstandsdiagram

På figur 3.49 ses tilstandsdiagrammet for use case 3. Diagrammet består af forskellige tilstande, hvor nogle er noteret med "Activity" for at vise, at konteksten i disse tilfælde er en Android aktivitet.



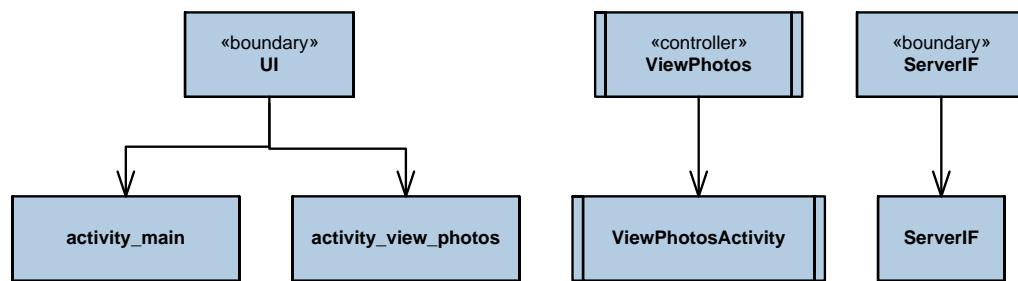
Figur 3.49. Tilstandsmaskine for use case 3.

3.3.4.2 Klassediagrammer

Ved analyse af af applikationsmodellen for use case 3 resulterer iteration 4 i følgende klassediagram. Den implementerede software til iterationen, er bygget op omkring dette klassediagram, og eventuelle afvigelser fra den implementerede kode er kun opdateret i klassediagrammet og altså ikke i sekvens- og tilstandsdigrammerne. I dette tilfælde er det kun app'ens klassediagram der påvirkes af iterationen, hvorfor det kun er dette klassediagram der opdateres. De klasser der blev identificeret i iteration 1, 2 og 3, er blevet opdateret, hvis der er fundet nye metoder eller attributter. De opdaterede klasser, og eventuelle nye klasser der er kommet til i iterationen, er fremhævet.

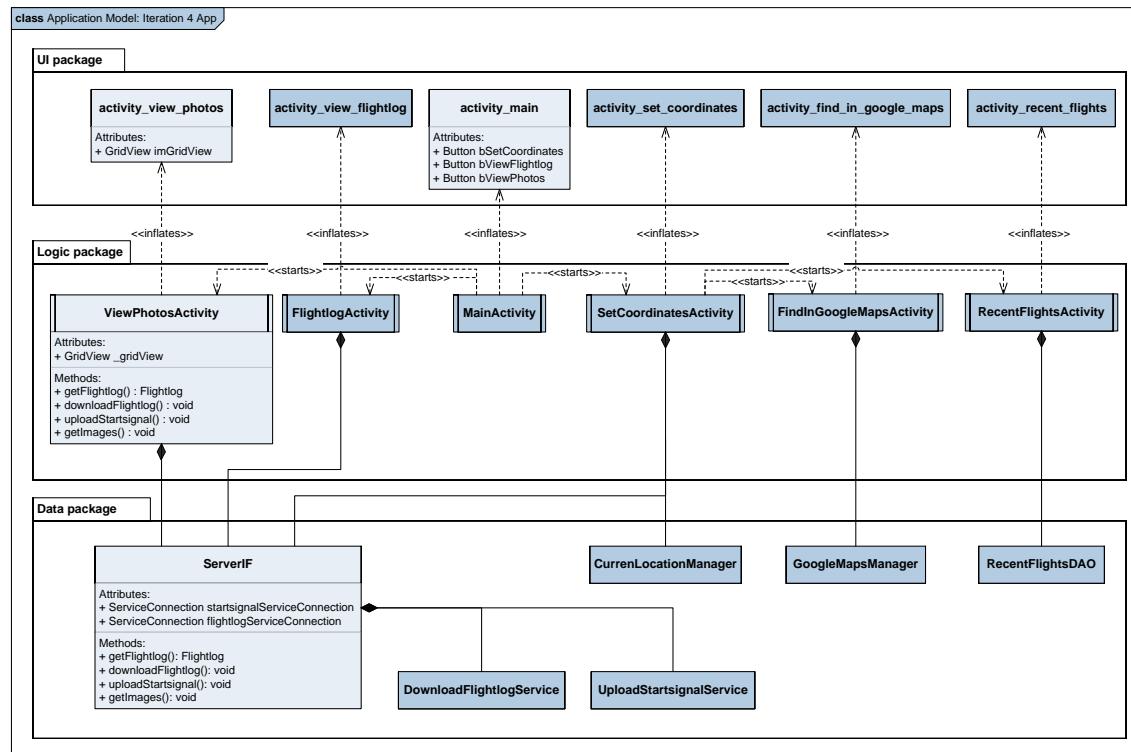
Klassediagram for app: Iteration 4

På figur 3.50 ses de softwareklasser, der er identificeret ud fra applikationsklasserne i applikationsmodellen for app'en.

**Figur 3.50.** Softwareklasser for app: Iteration 4

Der er tilføjet funktionalitet til den eksisterende softwareklasse *ServerIF*. Funktionaliteten i controller-klassen *ViewPhotos* varetages af softwareklassen *ViewPhotosActivity*. Samtidig er der tilføjet en ny UI-klasse til at indeholde UI-elementer for *ViewPhotosActivity*, mens der er tilføjet UI-elementer til den allerede eksisterende UI-klasse *activity_main*.

På figur 3.51 ses klassediagrammet for app'en i iteration 4. Udover identificerede klasser fra applikationsmodellen er der tilføjet hjælpeklasser til klassediagrammet for at isolere funktionalitet, der benyttes flere steder i systemet, og for at uddeletere ansvar som de øvrige klasser ellers skulle have haft.

**Figur 3.51.** Klassediagram for app: Iteration 4.

Metoder og attributter er udeladt i de klasser, hvor intet er ændret i forhold til forrige iterationer.

Tilføjede klasser i denne iteration:

- **activity_view_photos:**

Denne klasse indeholder de UI-elementer brugeren præsenteres for, i dette tilfælde et gitter-vindue til at vise billederne fra serveren.

- **ViewPhotosActivity:**

Denne klasse varetager de logiske operationer, og står for at igangsætte download af billeder fra server, så snart aktiviteten åbnes. Klassen står samtidig for at populere gitter-vinduet med billeder, så snart de er downloaded fra serveren.

Opdaterede klasser i denne iteration:

- **activity_main:**

Opdateret med knap til at åbne aktiviteten *ViewPhotosActivity*.

- **ServerIF:**

Denne klasse er opdateret med en *getImages* funktion, der står for at starte en baggrundstråd til download af en liste over de billeder, der er på serveren.

Klassediagram for drone: Iteration 4

Der er ikke fundet nye klasser, metoder eller attributter for dronen i denne iteration. Se derfor figur 3.46

Klassediagram for server: Iteration 4

Der er ikke fundet nye klasser, metoder eller attributter for serveren i denne iteration. Se derfor figur 3.33.

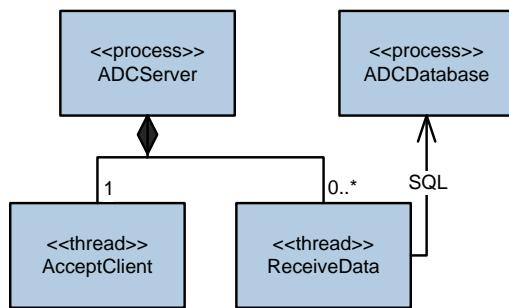
Herved konkluderes logical view'et, hvor der er givet en detaljeret gennemgang af designet af de enkelte use cases. Hver iteration er resulteret i de klassediagrammer, der er benyttet i softwaren.

3.4 Process view

Dette afsnit beskriver systemets processer og interaktionen mellem dem. Først gennemgås hver af de tre systemdele, server, app og drone hver for sig, og derefter gives et overblik over det samlede system.

3.4.1 Server

I dette afsnit gennemgås de interne processer og tråde, der kører på serveren, og kommunikationen mellem dem. På figur 3.52 ses et overblik over disse processer og tråde.



Figur 3.52. Processer og tråde på serveren.

Serveren og databasen kører i hver sin proces. Serveren opretter yderligere tråde, for at kunne håndtere flere klienter på samme tid. Nedenfor beskrives hver enkelt proces og tråd.

3.4.1.1 Processer

ADCServer

Dette er serverprocessen, der står for oprettelse af server-socket'en, og starter lytning på dens tildelte port. Processen står for at starte tråden *AcceptClient*, der accepterer forbindende klienter.

ADCDatabase

Databasen er placeret på en SQL server hvis proces kører sideløbende med *ADCServer*-processen. Databasen tilgås fra tråden *ReceiveData* gennem en SQL-forbindelse.

3.4.1.2 Tråde

Serverprogrammet er struktureret således, at .NET står for al trådhåndtering i forbindelse med den asynkrone socket-programmering. For f.eks. at lytte efter klienter på en socket

uden at blokere processen, benyttes et asynkront funktionskald fra et C# socket-objekt. .NET benytter tråde fra systemets thread pool⁶ til at udføre opgaven runtime. Derfor repræsenterer trådblokken på diagrammet ikke direkte enkeltstående tråde. De kunne i teorien være den samme tråd – det er op til systemets scheduler. Der henvises dog til dem som enkeltstående tråde, da de er opgaver, som kan udføres parallelt med hovedprocessen og andre opgaver.

AcceptClient

Denne tråd står for at acceptere en indkommende klient på server-socket'en. Når en klient forbindes, tilføjes klienten til en klientpulje, der indeholder aktive klient-sockets. Der startes en ny tråd, *ReceiveData*, der står for at modtage data fra den forbindende klient. Herefter gøres *AcceptClient*-tråden klar til igen at kunne acceptere en indkommende klient på server-socket'en. Med dette design startes en ny tråd, *ReceiveData*, for hver forbindende klient - dvs. at serveren kan håndtere flere klienter ad gangen, da hver klientforespørgsel håndteres i sin egen tråd.

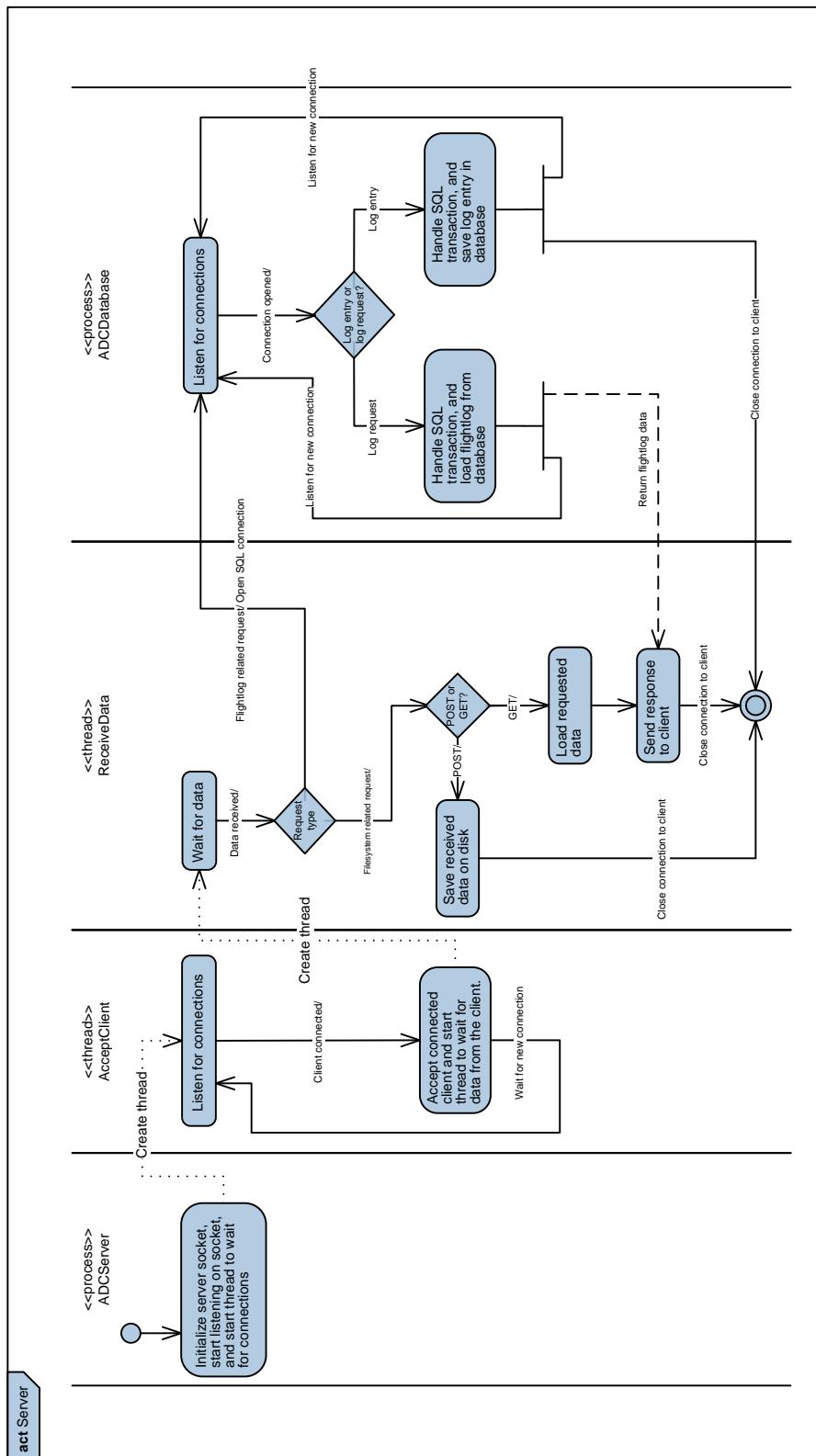
ReceiveData

Denne tråd venter på data fra den forbindende klient. Data modtages fra klienten i form af en HTTP-forespørgsel. Så snart data modtages, håndteres HTTP-forespørgslen i denne tråd. Hvis der forespørges på en logindførsel eller udlæsning, oprettes en SQL-forbindelse til databasen, og transaktionen gennemføres. I denne tråd sendes der samtidig svar tilbage til klienten, hvis dette er påkrævet. Efter forespørgslen er håndteret, lukkes klient-socket'en, og klienten fjernes fra klientpuljen.

3.4.1.3 Runtime kommunikation

Her ses et aktivitetsdiagram for den interne kommunikation mellem de forskellige processer og tråde på serveren.

⁶<http://msdn.microsoft.com/en-us/library/vstudio/5w7b7x5f%28v=vs.100%29.aspx>

*Figur 3.53.* Aktivitetsdiagram for serveren.

3.4.2 Drone

I dette afsnit gennemgås de interne processer og tråde, der kører på dronen, og kommunikationen imellem dem. På figur 3.54 ses et overblik over disse processer og tråde.



Figur 3.54. Processer og tråde på dronen.

På dronen kører tre éntrådede processer på de tre micro controllere, der sidder på dronen.

3G/GPS Moduler

Denne proces kører på 3G/GPS shield'et, der er forbundet med Arduino-board'et. Processen står for indhentning af GPS-data fra GPS-satellitter og internetkommunikation.

Main control

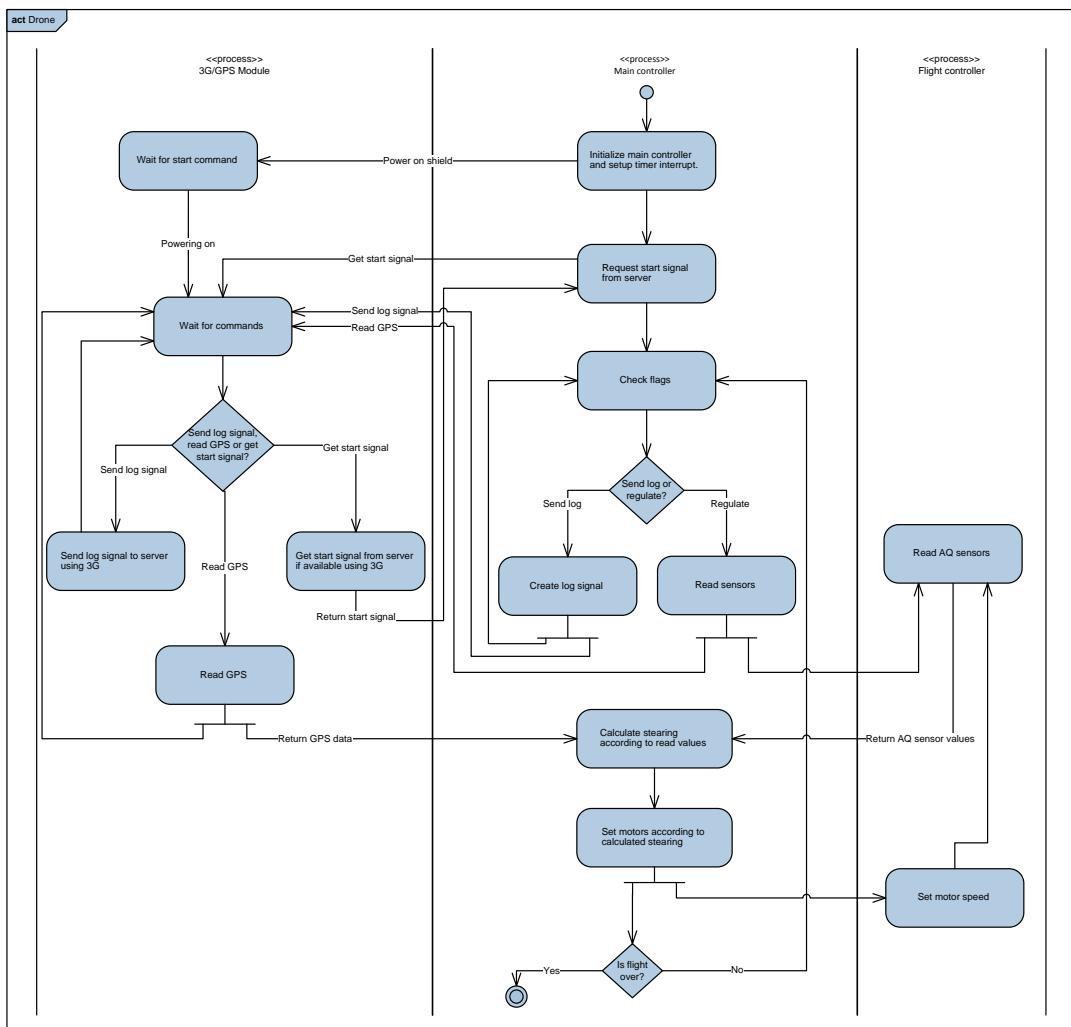
Denne proces kører på Arduino board'et, og er hoved-controlleren på dronen. Processen står for al indhentning af sensorværdier fra flight control board'et, indhentning af GPS-data samt kommunikation med serveren gennem 3G/GPS-modulet. Samtidig står processen for styring af dronen.

Flight control

Denne proces kører på Aeroquad board'et på dronen, og står for måling af sensorværdier og motorstyring.

3.4.2.1 Runtime kommunikation

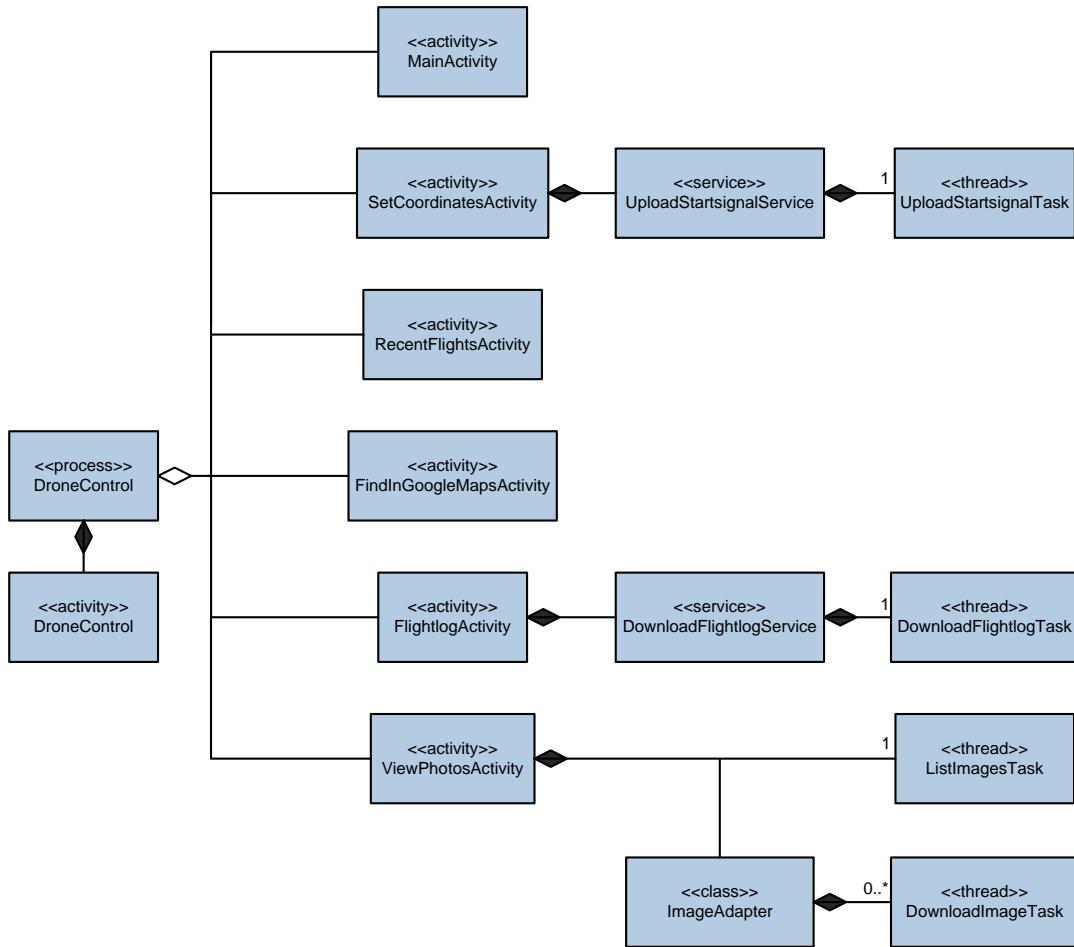
På figur 3.55 ses et aktivitetsdiagram for den interne kommunikation mellem de forskellige processer og tråde på dronen. Aktivitetsdiagrammet beskriver forløbet fra dronen tændes, og henter et startsignal fra serveren, til dronen er hjemvendt fra destinationen, og flyvningen er overstået.



Figur 3.55. Aktivitetsdiagram for dronen.

3.4.3 App

I dette afsnit gennemgås de interne processer og tråde, der kører på app'en. I Android eksekveres app'en i én proces med én hovedtråd, og aktiviteter i app'en eksekveres på denne tråd. Services eksekveres også på hovedtråden, men er ikke bundet til UI'et, som aktiviteter. *AsyncTask*'s bliver afviklet på baggrundstråde, og bliver i dette projekt benyttet, så snart der skal kommunikeres med serveren. På figur 3.56 ses et overblik over de processer, aktiviteter, services og tråde, app'en indeholder.



Figur 3.56. Processer og tråde i app'en.

3.4.3.1 Prossesser

DroneControl

Dette er processen, hvorpå Drone Control app'en kører. *DroneControl* udgør samtidig hovedtråden, hvorpå aktiviteterne i app'en bliver eksekveret.

3.4.3.2 Aktiviteter

DroneControl

Dette er den første aktivitet, der bliver startet op, når app'en startes. Aktiviteten står for at vise en opstartskærm for derefter at starte *MainActivity*.

MainActivity

Dette er hovedskærmen i app'en, der præsenterer brugeren for de overordnede funktioner app'en tilbyder i form af knapper.

SetCoordinatesActivity

Denne aktivitet giver brugeren mulighed for at vælge de koordinater, der skal sendes i startsignalet til dronen, og bliver startet af *MainActivity* ved tryk på knappen "Set Coordinates".

RecentFlightsActivity

Denne aktivitet giver brugeren et overblik over de seneste startsignaler, der er sendt til dronen fra app'en. Aktiviteten startes af *SetCoordinatesActivity* ved tryk på knappen "Recent Flights".

FindInGoogleMapsActivity

Denne aktivitet giver brugeren mulighed for at vælge et punkt på et Google Maps-kort. Aktiviteten startes af *SetCoordinatesActivity* ved tryk på knappen "Find in Google Maps".

FlightlogActivity

Aktiviteten står for at vise flight log'en hentet fra dronen. Aktiviteten startes af *MainActivity* ved tryk på knappen "Show Flightlog".

ViewPhotosActivity

Denne aktivitet står for at vise billeder hentet fra dronen i et gittervindue. Aktiviteten startes af *MainActivity* ved tryk på knappen "View photos".

3.4.3.3 Services

UploadStartsignalService

Denne service startes af *SetCoordinatesActivity*, efter at brugeren har accepteret aktivering af aktuelle koordinater. Servicen står for at starte upload af startsignal til dronen.

DownloadFlightlogService

Denne service startes af *FlightlogActivity*, når *FlightlogActivity* har gjort sin grafiske grænseflade klar til at vise flight log-data. Servicen står for at starte download af flight log'en fra dronen.

3.4.3.4 Tråde

Følgende tråde repræsenterer opgaver, der eksekveres på baggrundstråde i Android styresystemet, dvs. tråde der ikke blokerer den aktivitet eller service, de blev startet fra.

UploadStartsignalTask

Denne tråd startes af *UploadStartsignalService*, og står for at uploadet et startsignal med de valgte koordinater i *SetCoordinatesActivity*.

DownloadFlightlogTask

Denne tråd startes af *DownloadFlightlogService*, og står for at downloade og deserialisere flight log'en fra dronen.

ListImagesTask

Denne tråd startes af *ViewPhotosActivity*, og står for at hente adresserne på de billeder, der ligger på serveren.

DownloadImageTask

Denne tråd startes af *ViewPhotosActivity*, når dennes grafiske grænseflade er klar til at vise billeder hentet fra dronen. Billederne bliver hentet enkeltvis i hver sin tråd, hvorfor der på figur 3.56 er angivet et ukendt antal for tråden. Tråden står for at hente billederne enkeltvis og indsætte dem i et gittervindue i *ViewPhotosActivity*.

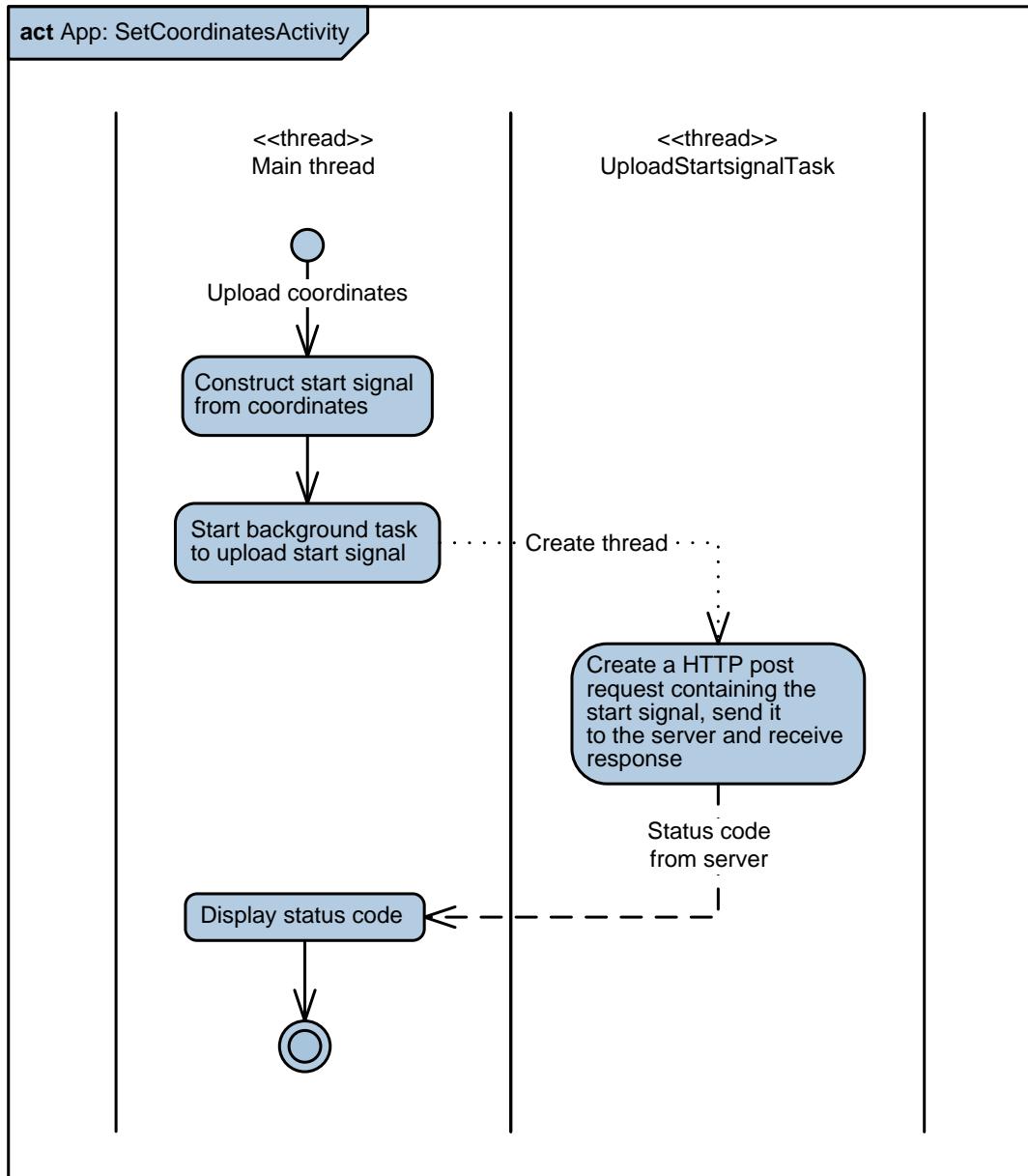
3.4.3.5 Runtime kommunikation

Kommunikationen mellem de forskellige tråde vil her blive beskrevet ved hjælp af aktivitetsdiagrammer. Der er konstrueret aktivitetsdiagrammer for de aktiviteter, hvori der kommunikeres med serveren. Dette er de eneste steder i app'en, hvor der indgår multithreading i designet, og vil derfor være de eneste steder, hvor det er aktuelt, at beskrive interaktion mellem tråde. Det skal noteres, at tråden, kaldet main thread, udgør

både aktiviteter og services, da disse afvikles på hovedtråden i Android, som beskrevet i afsnit 3.4.3.

SetCoordinatesActivity

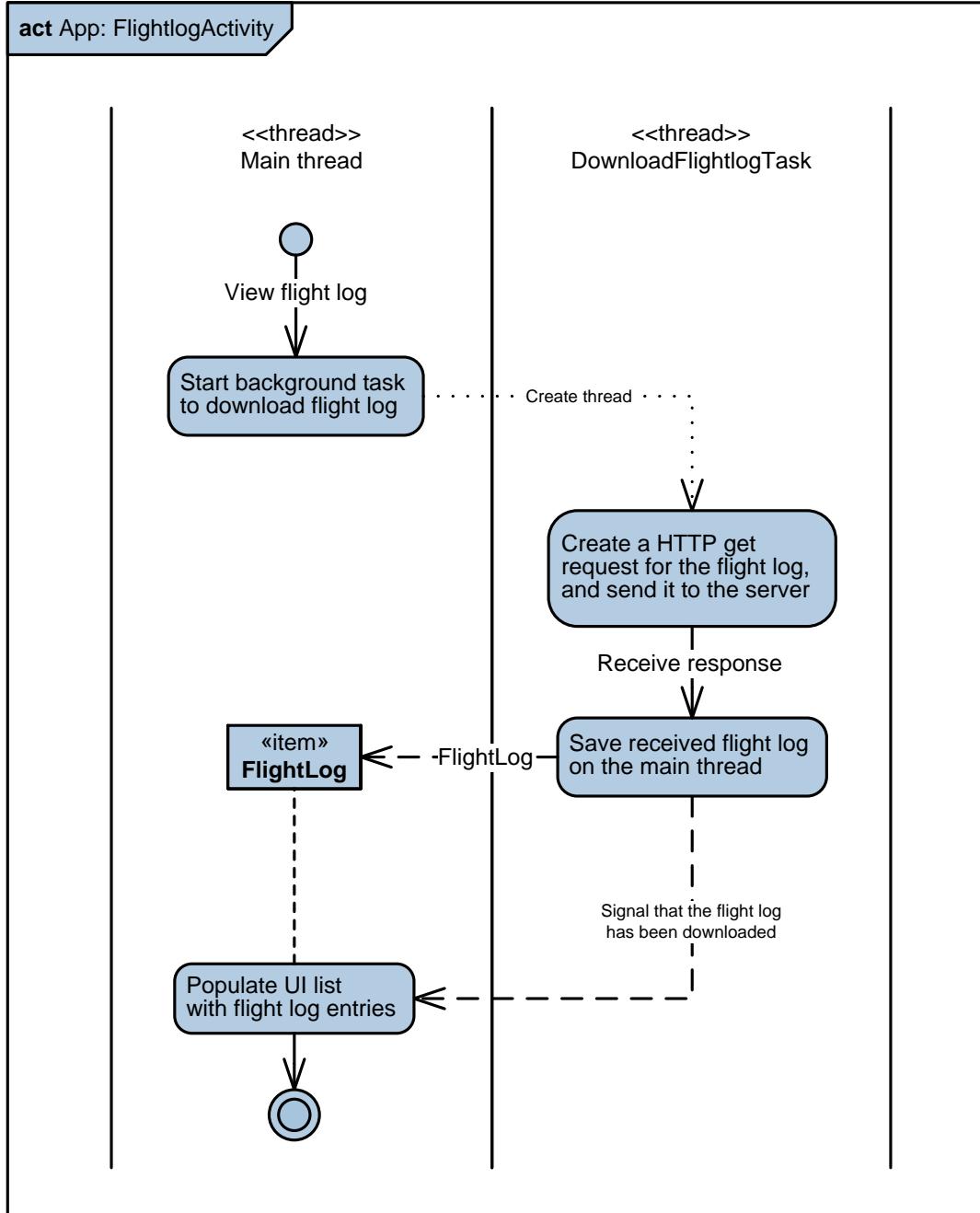
I aktiviteten *SetCoordinatesActivity* kan brugeren upload et startsignal til serveren med brugerdefinerede koordinater. På figur 3.57 ses et aktivitetsdiagram for program-flow'et, efter brugeren har valgt at overføre startsignalet, da det er her multithreading'en påbegyndes.



Figur 3.57. Aktivitetsdiagram for *SetCoordinatesActivity*.

FlightlogActivity

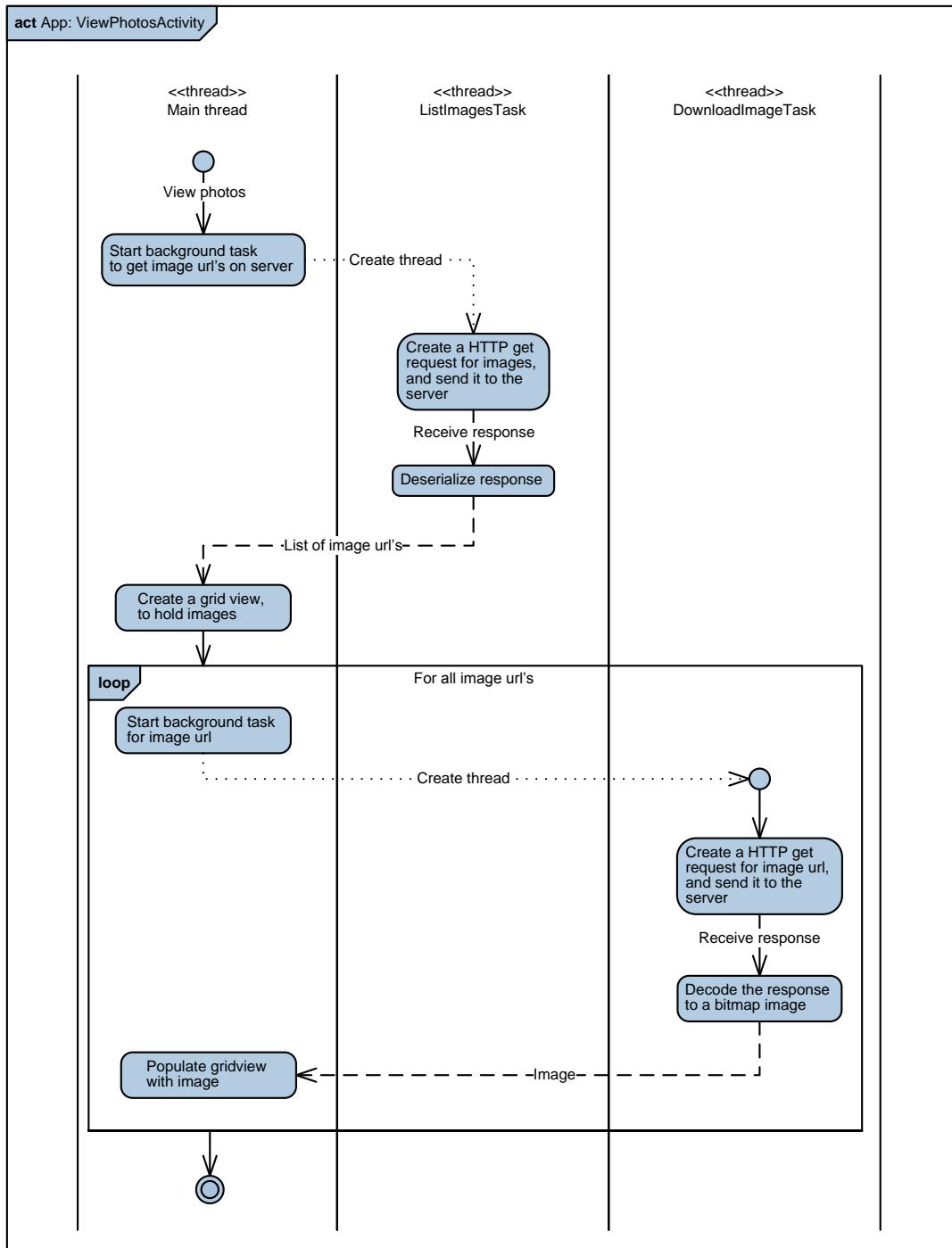
I denne aktivitet hentes flight log'en fra serveren, og vises i en liste. Her er lavet et aktivitetsdiagram for program-flow'et, lige efter brugeren har åbnet aktiviteten, da det er her multithreading'en påbegyndes.



Figur 3.58. Aktivitetsdiagram for *FlightlogActivity*.

ViewPhotosActivity

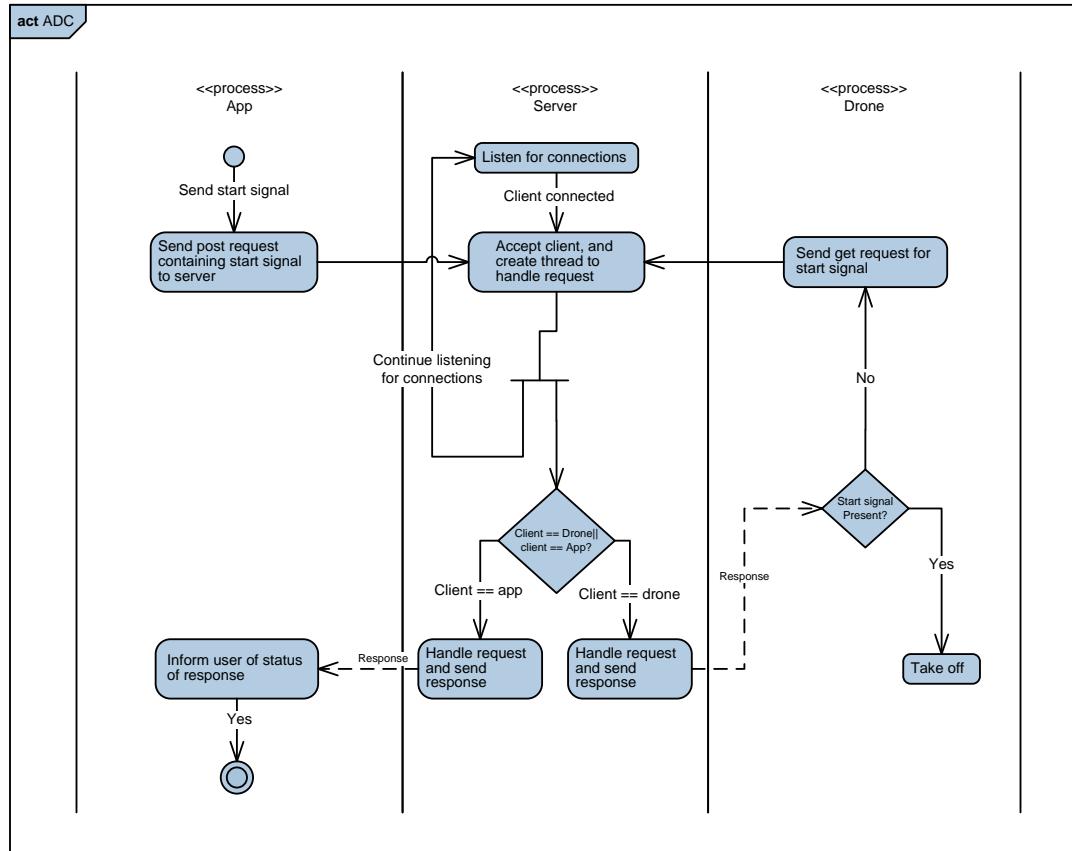
I denne aktivitet hentes billeder fra serveren, og vises i et gitter vindue. Her er lavet et aktivitetsdiagram for program-flow'et, lige efter brugeren har åbnet aktiviteten, da det er her multithreading'en påbegyndes.



Figur 3.59. Aktivitetsdiagram for `ViewPhotosActivity`.

3.4.4 Runtime kommunikation for det samlede system

På figur 3.60 ses et aktivitetsdiagram for et sammendrag af use case 2, 11, 12 og 13. Brugeren uploader startsignal til serveren, alt imens serveren lytter efter klienter, og dronen efterspørger startsignal på serveren. Her ses de tre systemdeler drone, server og app, som tre overordnede processer. Dette sammendrag af use cases tydeliggør kommunikationen mellem alle systemets overordnede processer, mens de alle er aktive.



Figur 3.60. Aktivitetsdiagram for det samlede system.

3.5 Data view

I dette afsnit beskrives det persistente data, der forefindes i systemet. Her er der både tale om data der gemmes på serveren og app'en.

3.5.1 Server

3.5.1.1 Database

På serveren gemmes flight log'en fra dronen i en database på en Microsoft SQL server⁷. Datatyperne, der benyttes til at gemme værdier i denne database, er:

- integer
- nchar

Integers benyttes til database-værdier, hvor heltal er tilstrækkelige til at rumme informationen. Til at gemme dato, tid og koordinater benyttes typen nchar, der kan indeholde unicode-strenge, hvilket medfører at hver karakter i strengen fylder to bytes. Ved at benytte denne datatype, er der mulighed for at gemme data i mange forskellige typer formater. Datatypen nchar kan kun indeholde strenge med en fast længde, der defineres, når databasen oprettes. Tabel 3.1 viser databasens flight log-tabel og dens datatyper.

Id	Date	Time	Longitude	Latitude	Heading	Status
integer	nchar(8)	nchar(6)	nchar(9)	nchar(9)	integer	integer

Tabel 3.1. Flight log-tabel.

Hver række i tabellen udgør en logpost fra dronen under flyvning, og indeholder oplysninger om dronens nuværende status.

Id

Hver række i tabellen får sit eget unikke id. Dette id bliver automatisk inkrementeret, hver gang der indsættes en ny logpost i flight log'en. Id-kolonnen er af datatypen integer. Den første logpost der indsættes, får tildelt id-værdien 1, den næste 2, osv.

Date

"Date"-kolonnen indeholder datoen for logposten. For at understøtte flest mulige dato-formater, benyttes datatypen nchar i denne kolonne. Længden af strengen er sat til otte karakterer for at understøtte formatet: "dd-mm-yy".

Time

"Time"-kolonnen indeholder tiden for logposten. Tidsformatet, der benyttes, er "hhmmss". For at understøtte flere tidsformater, til eventuel videreudvikling, benyttes igen datatypen nchar, så formatet let kan ændres.

⁷http://en.wikipedia.org/wiki/Microsoft_SQL_Server

Longitude og Latitude

Dronens nuværende koordinatsæt gemmes også som nchar, for let at kunne ændre formatet i eventuel videreudvikling. Det nuværende format er decimalgrader med seks decimaler.

Heading

For at vide hvilken retning dronen har i hver enkelt logpost, modtages også en værdi for dronens kurs. Denne værdi gemmes i "Heading"-kolonnen i databasen. Værdier for kurserne kan ligge mellem 0 og 360. Opløsningen af denne værdi er ikke kritisk, for at vide hvilken retning dronen har. Derfor benyttes datatypen integer, så kurserne kun repræsenteres af heltal.

Status

Med hver enkelt logpost sender dronen en status-variabel, der fortæller hvilken tilstand dronen er i på det pågældende tidspunkt. Denne værdi gemmes i "Status"-kolonnen i databasen, og repræsenteres af et heltal fra et til seks.

Alle databasens værdier, undtagen "Id"-kolonnen, sendes med, når app'en forespørger flight log'en. I app'en gengives databaseværdierne, som de er i databasen, bortset fra "Status-værdien, der omdannes til en tekst-streng, der beskriver dronens status.

3.5.1.2 Filsystem

Serveren gemmer startsignalet i en fil i serverens filesystem. Startsignalet gemmes som en xml-fil indeholdende startsignalet som en XML-formateret tekststreng. Startsignalet gemmes med XML-formatet vist i afsnit 3.6.3 Beskrivelse af dataformat i kapitel 3.6 Deployment View.

3.5.2 App

I app'en benyttes en SQLite database til at gemme de startsignaler, man vælger at sende til dronen. Samtidig gemmes de koordinater, der indskrives i koordinattekstfelterne på skærmen, hvor der vælges koordinater. Disse koordinater gemmes lokalt i app'en og medvirker, at brugeren ikke skal skrive koordinaterne igen, hvis der navigeres væk fra skærmen, eller app'en lukkes ned. Koordinaterne gemmes ved hjælp af Android-klassen *SharedPreferences*. Disse to datalagringstyper beskrives nærmere i dette afsnit.

3.5.2.1 Database

Den eneste datatype der benyttes i app'ens database, er strings. Tabel 3.2 viser databasens "Recent Flights"-tabel.

Id	Date	Longitude	Latitude
string	string	string	string

Tabel 3.2. "Recent Flights"-tabel.

Id

Denne kolonne indeholder, ligesom på serveren, et unikt id, der tildeles hver række i databasen.

Date

Denne kolonne indeholder dato og tid for, hvornår startsignalet blev uploadet. Formatet er "dd/mm/yyy hh:mm:ss".

Longitude* og *Latitude

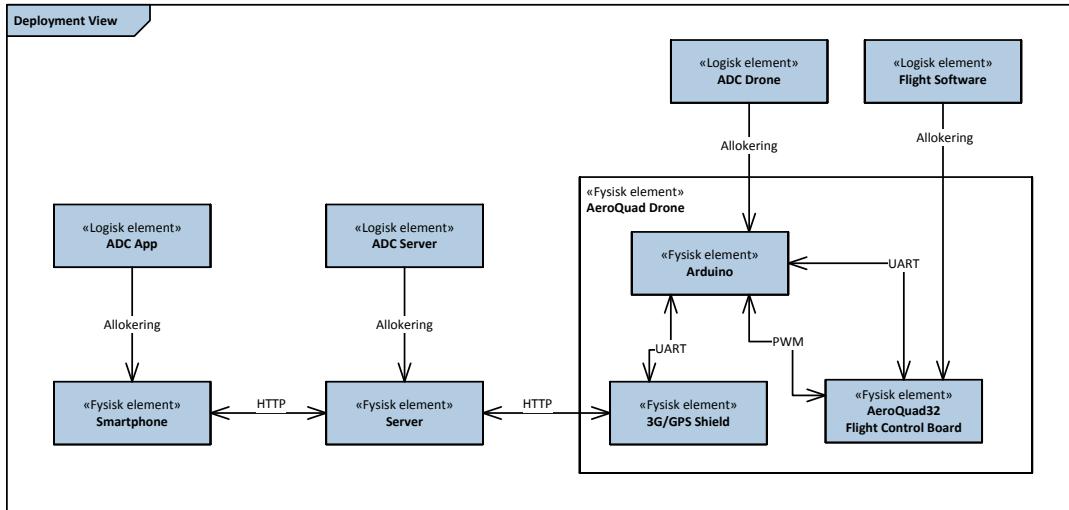
Denne kolonne indeholder koordinatsættet som dronen skal flyve til. Koordinatsættet gemmes i samme format som på serveren; decimalgrader med seks decimaler.

3.5.2.2 Shared Preferences

Med *SharedPreferences* er det muligt at gemme data i app'ens interne filsystem på telefonen. Det benyttes, når brugeren har indtastet koordinater i koordinattekstfelterne i *SetCoordinatesActivity*, hvor der vælges koordinater, som skal uploades til serveren. Så snart brugeren forlader denne aktivitet, vil de koordinater, der står i koordinattekstfelterne, blive gemt i *SharedPreferences*. Når brugeren åbner aktiviteten igen, vil koordinaterne blive hentet fra *SharedPreferences*, og indskrevet i koordinattekstfelterne igen. Koordinatsættet gemmes som to tekststrenge.

3.6 Deployment View

I dette afsnit vil det blive beskrevet, hvordan de forskellige dele af systemet er implementeret på forskellige platforme. Derudover vil kommunikationen og enkelte protokoller blive gennemgået.



Figur 3.61. Systemets software allokeret på fysiske enheder.

På figur 3.61 ses hele systemet. Der er ingen direkte kommunikation mellem drone og smartphone. Her er serveren bidelede mellem de to, og står for kommunikationen mellem dem. Til at kommunikere med serveren anvendes 3. generations mobilnetværk. Serveren benytter TCP som transportlag og modtager og sender data via HTTP-protokollen⁸. Den indbyrdes kommunikation på dronen er seriell kommunikation via UART.

På figur 3.61 ses det også, hvordan de logiske elementer er allokeret på fysiske enheder.

3.6.1 Allokering

3.6.1.1 Drone

Logikken på dronen består af 3 dele. Til at kommunikere med serveren via 3G er der anvendt et shield. Dette shield anvendes også til at få GPS-information, og er desuden tiltænkt opgaven at tage billeder samt at uploadne disse til serveren.

Til at regulere motorerne anvendes et AeroQuad32 Flight Control Board (AQ board). Udeover motorregulering indeholder AQ boardet sensorer, der monitoreres. Der er ikke skrevet noget kode til denne enhed, men der er derimod ændret i dens source kode. Denne ændring medfører, at den serielle kommunikation med board'et foregår via UART i stedet for USB. Ändringen er nødvendig for at kunne læse sensorværdierne fra Arduino'en.

⁸http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Arduino'en er implementeret som "hovedcomputer" på dronen. Den styrer, hvornår shield'et skal kommunikere med serveren, og hvordan AQ board'et skal styre motorerne ved hjælp af PWM signal.

I tabel 3.3 ses en oversigt over de interne forbindelser på dronen.

UART	
Baud rate	115200 bits/s.
Character framing	Otte databit, nul paritet og ét stop bit.
PWM	
Spænding	0-5V
Frekvens	50Hz
Duty-cycle	Mellem 5-10%.

Tabel 3.3. Beskrivelse af interne forbindelser på dronen.

3.6.1.2 Smartphone

Applikationen er implementeret til at køre på en smartphone med Android version 4.0 (Ice Cream Sandwich) eller nyere. Derudover skal det være muligt at gå på nettet og tilgå GPS. Såfremt disse krav imødekommes, vil enhver Android baseret smartphone kunne anvendes.

3.6.1.3 Server

Serveren er udviklet som en C#-konsolapplikation og benytter en Microsoft SQL server⁹. Serveren skal derfor eksekveres på en Windows-baseret maskine.

3.6.2 Beskrivelse af protokol

Som kommunikation mellem server og klienter i systemet (app og drone) anvendes HTTP-protokollen. Denne protokol er request-repsonse baseret, hvilket betyder, at en klient sender en forespørgsel til serveren, hvorefter serveren sender et svar til klienten. Klienten kan enten forespørge på data eller sende data til serveren.

Både forespørgsel og svar er bygget op af en header og en body. Header'en indeholder oplysninger om afsenderen og om forespørgslens karakteristika, hvor body'en indeholder selve den data, der forespørges, eller sendes, til serveren.

I dette projekt benyttes HTTP-protokollen til kommunikation mellem de overordnede systemdeler. En vigtig egenskab eller begrænsning ved HTTP er, at det altid er klienten, der skal tage initiativet. Det er ikke muligt for serveren at foretage sig noget, uden klienten har sendt en forespørgsel. Det passer imidlertid med den måde, kommunikationen skal foregå i dette system.

⁹http://en.wikipedia.org/wiki/Microsoft_SQL_Server

I HTTP findes mange forskellige typer af forespørgsler herunder GET og POST, der anvendes i dette projekt. GET-forespørgslen er den mest anvendte indenfor HTTP, og anvendes i dette system, når dronen skal hente startsignal, eller når app'en ønsker at hente billeder eller logfiler.

Til at sende indhold til serveren anvendes POST. I denne forespørgsel medsendes data i body'en. Såfremt det medsendte data godkendes, kan det anvendes af serveren. I dette system anvendes POST til at sende startsignalet fra app'en samt logposter fra dronen.

3.6.3 Beskrivelse af dataformat

Når der sendes data mellem smartphone og drone, sker dette via serveren. Dog er det ikke altid lige simpelt at sende et objekt eller en datastruktur over en dataforbindelse. Derfor serialiseres data til et passende fladt format - i dette system anvendes XML. Mellem de forskellige enheder i projektet benyttes de samme datastrukturer. I dette delafsnit vil formatet af startsignaler og logsignaler blive beskrevet. De er begge XML-formaterede, og indeholder hovedsagligt de samme informationer.

Startsignal

Et startsignal sammensættes når brugeren trykker ”Aktiver Koordinater” i app'en. Herefter uploades signalet til serveren via en HTTP POST request. På serveren vil startsignalet blive gjort tilgængeligt for dronen. Her kan det downloades ved en HTTP GET request. Det startsignal der bliver dannet på smartphone'en, er opbygget på samme måde, som det dronen modtager. Startsignalet indeholder følgende:

- coordinates: Koordinatsæt for den ønskede destination, hhv. breddegrad og længdegrad.
- date: Dato i formatet ”DD-MM-YY”.
- time: Tid i formatet ”TTMMSS”.

Startsignalets XML-format ses herunder:

```
<startsignal>
  <coordinates>
    <latitude>32.9823</latitude>
    <longitude>10.1915082931519</longitude>
  </coordinates>
  <date>17-11-14</date>
  <time>144032</time>
</startsignal>
```

Dette signal indeholder de oplysninger, der er nødvendige, for at dronen kan flyve ud til destinationen.

Logsignal

Et logsignal sendes med et fast tidsinterval. Ved at sende dette signal opdateres flight log'en løbende med detaljer fra flyvningen. Det indeholder følgende:

- coordinates: Nuværende koordinatsæt, hhv. længdegrad og breddegrad.
- time: Tid i formatet "TTMMSS".
- date: Dato i formatet "DD-MM-YY".
- heading: Nuværende kurs målt i grader.
- status: Hvilken tilstand dronen er i (Landed, TakingOff, FlyingOut, AtDestination, FlyingHome, Landing, EmergencyLanding, EmergencyLanded).

Logsinalet er opbygget på samme måde som startsignalet: Som en XML-formateret tekststreng der modtages i samme format, som det afsendes. Herunder ses XML-formatet:

```
<log>
  <coordinates>
    <latitude>56.1719243</latitude>
    <longitude>10.1916175</longitude>
  </coordinates>
  <time>104200</time>
  <date>19-11-14</date>
  <heading>0</heading>
  <status>Landed</status>
</log>
```

3.6.4 Serialisering af data

Serialisering og deserialisering af data sker forskelligt, alt efter hvilken systemdel der er tale om. I dette afsnit beskrives, hvorledes serialisering og deserialisering foregår på de forskellige dele af systemet.

3.6.4.1 Drone

Deserialisering af startsignal

Ved modtagelse af startsignal på dronen kopieres selve body'en af HTTP response'et over i et char array, og er derefter klar til at blive deserialiseret. Hver variabel i XML objektet er omgivet af sit navn i form af et tag, `<starttag>VALUE</sluttag>`. Et eksempel kunne være tid:

```
<time>093605</time>
```

Det er derfor meget simpelt at kopiere variablenes værdi. Først findes pointeren til sluttegnet i start-tag'et for variablen. Her markeret med underline `<time>`. Dernæst findes pointeren til starttegnet i slut-tag'et: `</time>`. Til at finde pointerne kan funktionen `strstr`¹⁰ anvendes.

¹⁰<http://www.cplusplus.com/reference/cstring/strstr/>

Ved at finde differencen mellem pointerne fås antallet af tegn, som værdien fylder. Da man nu har antallet af tegn, kan funktionen `strncpy`¹¹ anvendes til at kopiere værdien, der er mellem de to pointere. Funktionen kopierer et angivet antal tegn fra et char array til et andet array. Hvis indholdet ikke blot er en tekststreng, kan funktionen `atof` anvendes til at konvertere en decimal tekststreng værdi til fx en double eller float. I dette tilfælde anvendes en struct til at opbevare de modtagne data. Det er derved væsentligt nemmere at anvende i systemet.

Serialisering af logsignal

Til afsendelse af logsignalet anvendes serialisering. Hver logpost indeholder de aktuelle GPS-koordinater, tid, dato, kurs samt dronens status. På dronen eksisterer det i klasser eller andre variabler. Det kan, som nævnt tidligere, ikke sendes råt, og skal derfor konverteres til et fladt format. Til afsendelse anvendes igen XML og det meste af XML-objektet er lavet på forhånd ved at allokerer char arrays, der indeholder start- og slut-tags. Ved at kopiere de aktuelle værdier ind i de manglende pladser mellem start og slut-tags'ene fås et char array klar til afsendelse. Dette array indeholder nu hele XML-objektet, og kan sendes til serveren.

3.6.4.2 Server

På serveren benyttes objekter til at indeholde de forskellige datastrukturer, der figurerer mellem de overordnede systemdelene. I kodeeksempel 3.1 ses *StartSignal*-objektet, der indeholder de navne og oplysninger, der serialiseres til en XML-formateret streng, før det afsendes.

Kodeeksempel 3.1. Startsignal- og Coordinates-objekt.

```
[Serializable()]
[XmlRoot("startsignal")]
public class StartSignal
{
    [XmlElement("coordinates", typeof(Coordinates))]
    public Coordinates coordinates { get; set; }

    [XmlElement("time")]
    public string Time { get; set; }

    [XmlElement("date")]
    public string Date { get; set; }
}

[Serializable()]
public class Coordinates
{
    [XmlElement("longitude")]
    public string Longitude { get; set; }
```

¹¹<http://www.cplusplus.com/reference/cstring/strncpy/>

```

[XmlElement("latitude")]
public string Latitude { get; set; }
}

```

Ligeledes findes der objekter for de andre datastrukturer, som serveren modtager fra klienter. Disse objekter kan vha. biblioteket System.Xml.Serialization¹² serialiseres til XML-formaterede strenge. Omvendt kan XML-formaterede strenge deserialiseres til disse objekter. Deserialisering og serialisering foregår i klassen *XmlHelper* på serveren og sker, når eksempelvis et XML-formateret startsignal er modtaget i en HTTP POST request, og skal gemmes i filsystemet. Først deserialiseres XML-strenge til et *StartSignal*-objekt for at verificere, at formatet af XML-strenge er korrekt. Hvis formatet af det modtagne startsignal ikke stemmer overens med *StartSignal*-objektet, vil deserialiseringen nemlig ikke kunne lade sig gøre. Efter en succesfuld deserialisering serialiseres startsignalet igen for dernæst at blive gemt i en fil i serverens filsystem.

Når der modtages en XML-formateret logpost i en HTTP POST request, deserialiseres XML-strenge til et *LogEntry*-objekt, som det ses på kodeeksempel 3.2.

Kodeeksempel 3.2. *Logentry*-objekt.

```

[Serializable()]
[XmlRoot("log")]
public class LogEntry
{
    [XmlElement("coordinates",typeof(Coordinates))]
    public Coordinates coordinates {get; set;}

    [XmlElement("time")]
    public string Time { get; set; }

    [XmlElement("date")]
    public string Date { get; set; }

    [XmlElement("heading")]
    public int Heading { get; set; }

    [XmlElement("status")]
    public int status { get; set; }
}

```

LogEntry-objektet gør også brug af *Coordinates*-objektet, ligesom *StartSignal*-objektet. Når dette *LogEntry*-objekt er erhvervet fra POST request'en, benyttes objektet til at indsætte de aktuelle værdier fra logposten i databasen. Den omvendte proces gennemgås, når der skal sendes en flight log til app'en, hvor der erhverves *LogEntry*-objekter for hver logpost tilstede i databasen, som indsættes i et *FlightLog*-objekt. *FlightLog*-objektet deserialiseres herefter, for at kunne sendes i et HTTP response til app'en.

¹²<http://msdn.microsoft.com/en-us/library/system.xml.serialization%28v=vs.110%29.aspx>

3.6.5 App

Serialisering og deserialisering i app'en foregår ligesom på serveren igennem objekter. I app'en benyttes biblioteket org.simpleframework.xml¹³ blot i stedet for det på serveren anvendte bibliotek System.Xml.Serialization.

I app'en modtages eksempelvis en flight log fra serveren som svar på en HTTP GET request, der serialiseres til et *FlightLog*-objekt, ligesom det der ses i kodeeksempel 3.3:

Kodeeksempel 3.3. FlightLog-objekt.

```
@Root  
public class FlightLog  
{  
    @ElementList  
    public List<FlightLogEntry> LogList;  
  
    public List getFlightLog() {  
        return LogList;  
    }  
}
```

Her har *FlightLogEntry* samme format som *LogEntry*-objektet på serveren. Efter serialiseringen til *Flightlog*-objektet, kan de aktuelle værdier let ekstraheres fra objektet, og benyttes til at populere UI-elementer som f.eks. en liste.

Der foretages ydermere serialisering og deserialisering, når app'en skal sende et startsignal til dronen, og når app'en skal hente billeder fra serveren.

Hermed konkluderes deployment view'et, hvor der er givet en detaljeret beskrivelse af logiske allokeringer på fysiske blokke i systemet, samt protokoller anvendt i kommunikationen mellem systemets forskellige underdele. Samtidig er de anvendte dataformater beskrevet.

¹³<http://simple.sourceforge.net/home.php>

3.7 Implementation view

Dette afsnit omhandler opsætning og installation af de forskellige softwareprojekter, således at det er muligt at fortsætte arbejdet med projektet. Det vil blive gennemgået, hvorledes de forskellige platformes udviklingsmiljøer opsættes, og gøres klar til brug.

Herunder vil de forskellige platformes filer blive overskueliggjort. Dette er gjort for at give et bedre overblik for en ny udefrakommende udvikler. Samtidig gennemgås softwareelementer og -klasser, der ikke er beskrevet i logical view, men som har betydning for de forskellige programmer, der afvikles på systemdelene.

Afsnittet er delt op i følgende underafsnit:

1 Opsætning og installation

- 1.1 Main controller (herunder AtmelStudio 6, MegunoLink samt Arduino filer og diverse drivere)
- 1.2 AeroQuad Flight Control Board v. 2 (AeroQuad Configurator samt diverse drivere)
- 1.3 Android Applikation (Eclipse med Android Development Tools plugin)
- 1.4 Server (Microsoft Visual Studio 2013)
- 1.5 Source-kode

2 Yderligere softwareelementer og -klasser

- 2.1 Motordriver
- 2.2 Arduino kode til C++
- 2.3 ADCDroneController state machine
- 2.4 Log state machine
- 2.5 PID-regulering
- 2.6 Geografiske beregninger

3.7.1 Opsætning og installation

3.7.1.1 Main controller

For at kunne udvikle til Arduino'en i dette projekt, skal udviklingsmiljøet opsættes. Først og fremmest skal man kunne overføre kode til Arduino'en. Ved installation af Arduino IDE'et fås et program, der hedder AVRdude. Dette program anvendes til at uploadede kode til Arduino'en. IDE'et indeholder også et omfattende bibliotek, der ønskes anvendt. De filer, der fås fra Arduino IDE'et, er vedlagt på CD'en, og derfor skal IDE'et kun installeres for at få de nødvendige drivere¹⁴. Derudover skal der installeres to andre programmer: MegunoLink og AtmelStudio 6. AVRdude vil blive brugt i programmet MegunoLink til at overføre kode via den serielle forbindelse. Arduino-biblioteket anvendes i AtmelStudio. Dette er en specialudgave af Microsoft Visual Studio, som Atmel har lavet til udvikling af kode til deres micro controllere. Disse to programmer er de eneste, der skal installeres og opsættes.

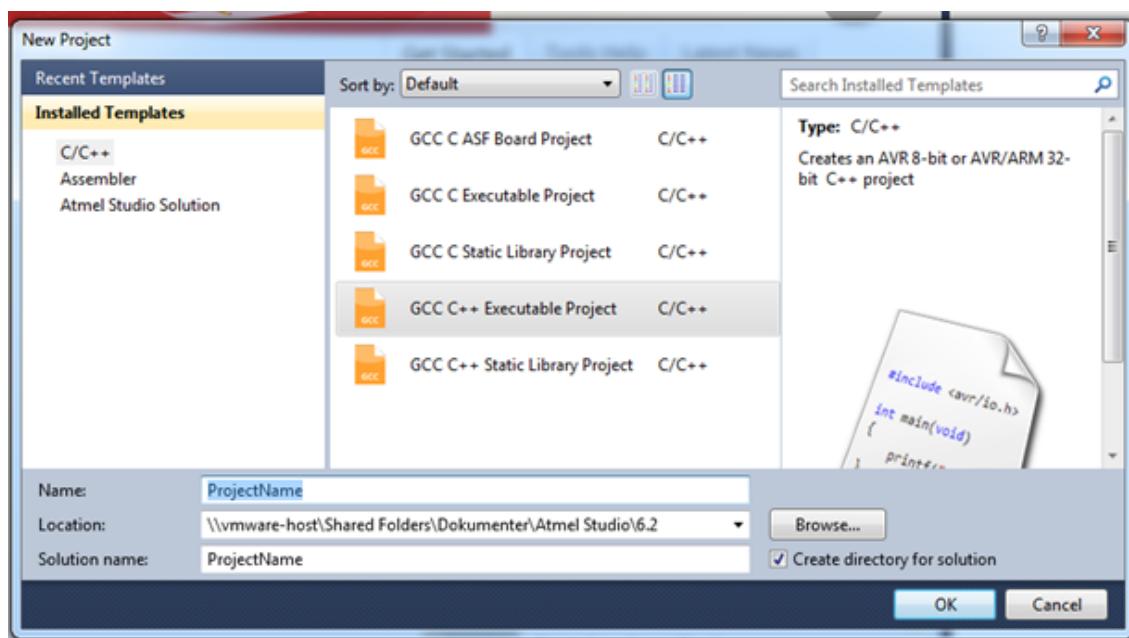
Guiden for main controlleren indeholder tre dele: Første del gennemgår opsætning af et nyt projekt i AtmelStudio. Anden del forklarer opsætning af MegunoLink. Tredje

¹⁴For yderlige information se følgende: Windows: <http://arduino.cc/en/Guide/Windows>. OSX:<http://arduino.cc/en/guide/macOSX>.

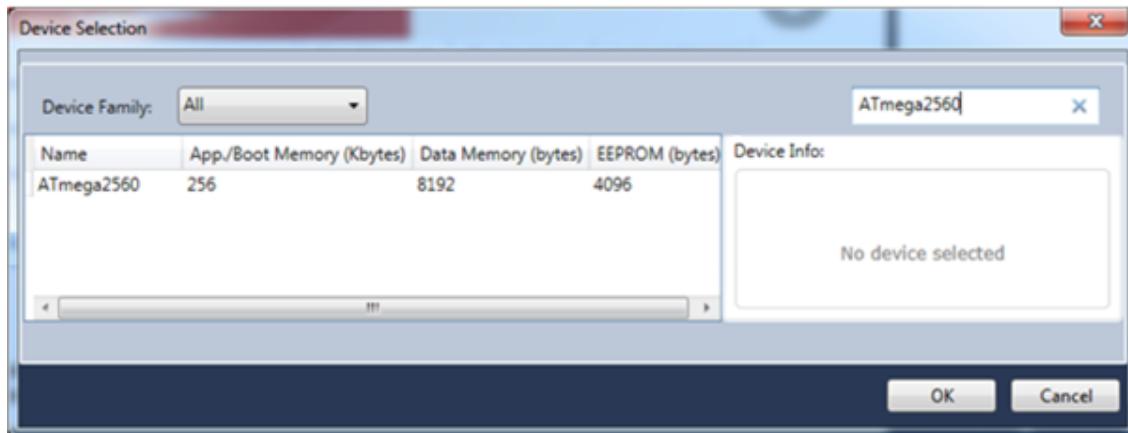
og sidste del indeholder import af kode fra projektet. Første del er vedlagt, hvis en udvikler ønsker at starte fra bunden af. Alle filer der er nødvendige, er vedlagt på CD'en, herunder biblioteker, header- og source-filer samt drivere. Alt dette er vedlagt, for at undgå kompatibilitetsproblemer med senere udgaver af Arduino-biblioteket. Det er nødvendigt at gennemføre del to, for at kunne overføre kode til Arduino Mega2560.

AtmelStudio 6

Installationen af AtmelStudio kræver ikke nogen speciel tilgang. Det er først ved oprettelse af et nyt projekt, at man skal være opmærksom. Derfor er følgende opsætningsprocedure lavet til oprettelse af et nyt AtmelStudio-projekt. Det er dog ikke nødvendigt at oprette et nyt projekt, da der, som nævnt tidligere, er vedlagt det færdige AtmelStudio software projekt. Dette projekt kan kopieres over til udviklerens AtmelStudio workspace og åbnes. Ved anvendelse af det vedlagte projekt bør versionen af AtmelStudio være 6.2.1153 men andre udgaver bør også fungere. **Nyt projekt i AtmelStudio 6.** Åben AtmelStudio 6 og vælg et "C++ Executable" projekt i "New Project" -wizard'en. Her vælges ATmega2560 som micro controller. Husk at give projektet et passende navn.



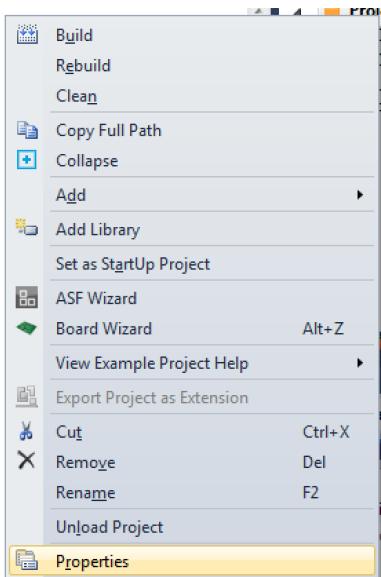
Figur 3.62. "New Project" -wizard. Vælg GCC C++ Executable Project.



Figur 3.63. Vælg ATmega2560. Det er muligt at søge øverst i højre hjørne.

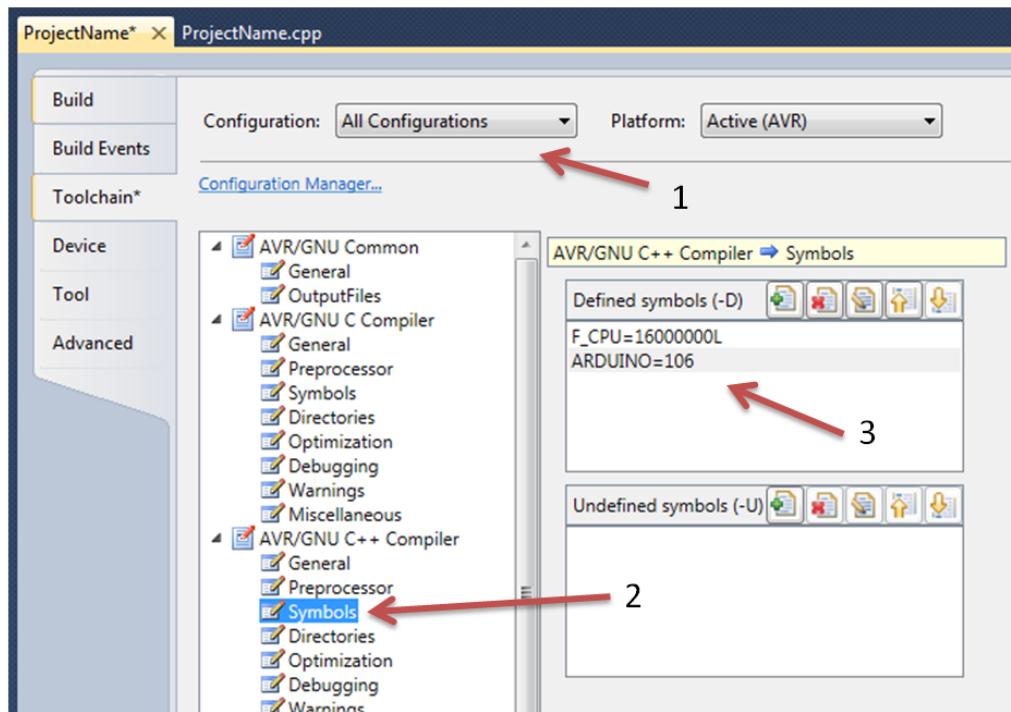
AtmelStudio opretter nu et tomt projekt. Før Arduino-biblioteket kan anvendes, skal det tilføjes i projektopsætningen. I hver fil man ønsker at anvende Arduino biblioteket skal Arduino.h inkluderes.

Start med at højreklikke på projektfilen og vælg "Properties".



Figur 3.64.

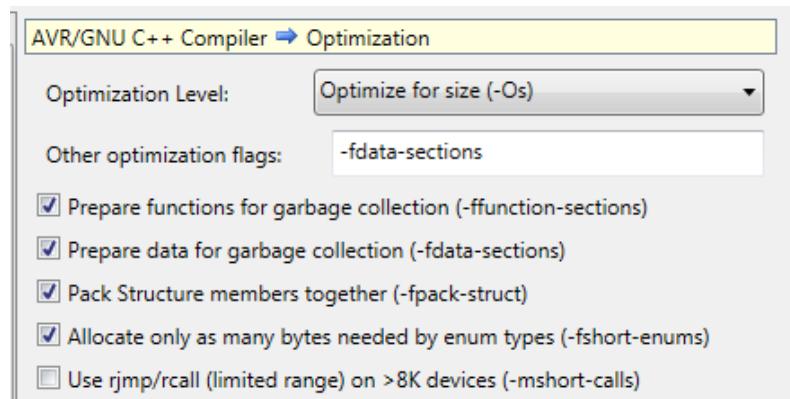
Vælg først "All Configurations" i dropdownmenyen "Configurations". Herefter vælges "Toolchain", og dernæst "Symbols" under "AVR/GNU C++ Compiler".



Figur 3.65.

Tilføj herefter ved ”Defined symbols” linjerne ”F_CPU=16000000L” samt ”ARDUINO=106”.

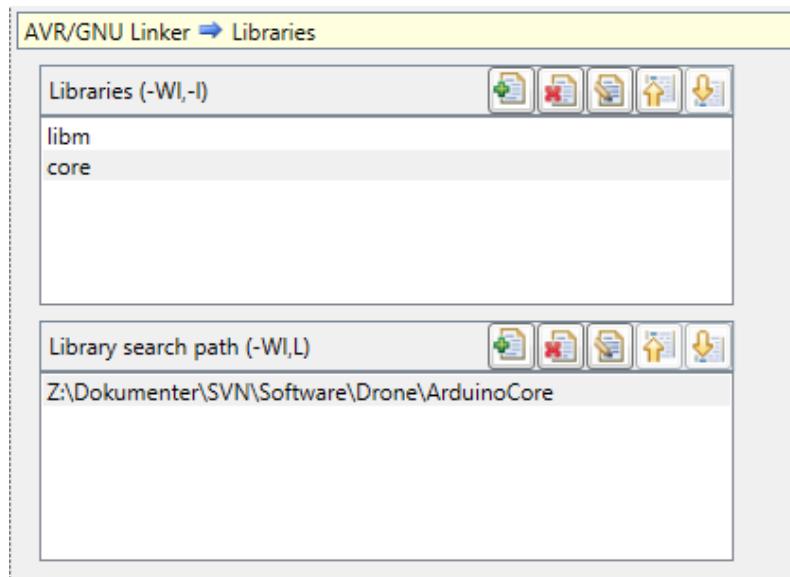
Gå herefter til ”Directories”. På CD’en er der en mappe ved navn ”ArduinoFiles”. I denne mappe ligger der to mapper hhv. ”arduino” og ”standard”. Disse mapper skal inkluderes i projektet. Kopier derfor mappen ”ArduinoFiles” over til stien ”..\\My Documents\\Atmel Studio\\” eller der hvor AtmelStudio lægger projektfiler. Klik på den grønne knap ”Add item” og tilføj stien til de to mapper. Sæt flueben ved ”Relativ path”. Gå herefter til ”Optimization”. Vælg ”Optimization level” til ”Optimize for size (-Os)” og tilføj ”-fdata-sections” til ”Optimization flags” tekstdokumenten.



Figur 3.66.

Til sidst skal der ændres i linker opsætningen. Under ”AVR/GNU Linker” vælges Libraries.

I boksen øverst tilføjes ”core”. På CD’en er filen ”Libcore.a” vedlagt. Gå til stien ”..\My Documents\Atmel Studio\”, eller der hvor AtmelStudio lægger sine projekter. Lav en ny mappe og kald den ”ArduinoCore” eller giv den et andet passende navn. Kopier herefter filen ”Libcore.a” ind i mappen. Denne fil er autogenereret fra Arduino IDE’et, og er lavet specifikt til en ATmega2560. Ved at inkludere denne fil er det muligt at bruge funktioner og klasser fra Arduino verdenen. I det nederste vindue tilføjes nu stien til denne mappe, der indeholder Arduino biblioteket.



Figur 3.67.

Vælg herefter punktet ”Optimization” og sæt flueben ved ”Garbage Collect unused sections (-Wl, -gc-sections)”. Dette sørger for, at ubenyttede dele af bibliotekerne ikke inkluderes i den endelig kodelinje. Altså en væsentlig reduktion i filstørrelse. I hoved *.cpp filen tilføjes nu linjen ”init();”. Dette er nødvendigt for at kunne benytte de Arduino specifikke ting. For at teste om det nye projekt fungerer kan følgende kode bruges til test. Kompilér ved at trykke F7 og AtmelStudio laver nu en *.hex fil. Denne fil skal overføres til Arduino’en vha. MegunoLink.

```
#include "Arduino.h"
#include <avr/io.h>

int main(void)
{
    init();

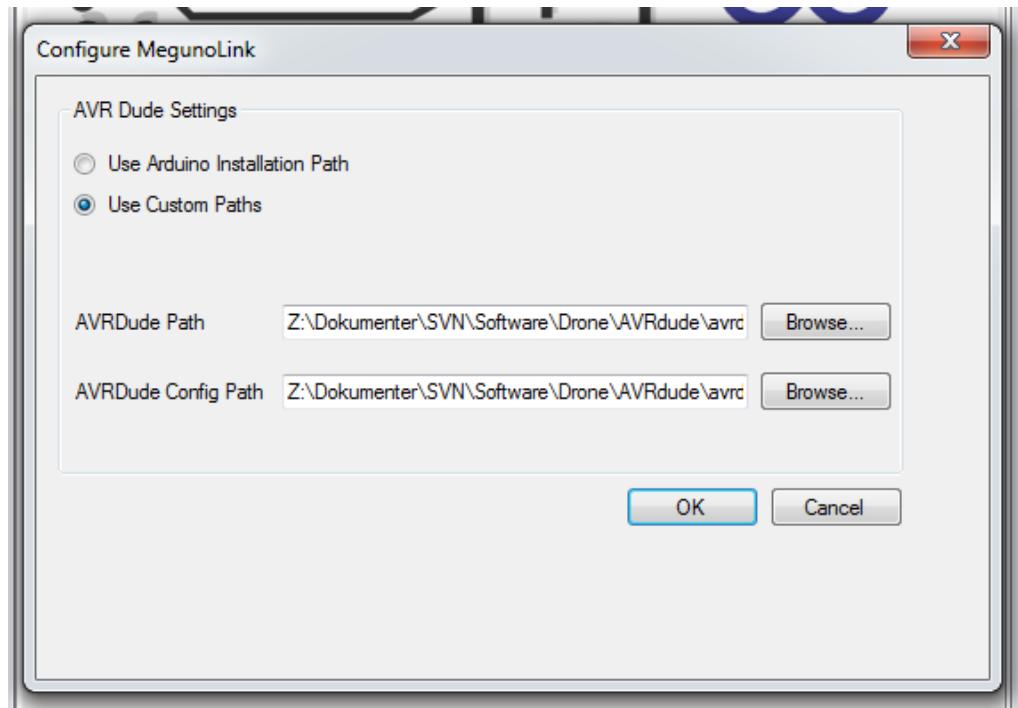
    Serial.begin(115200);
    while(1)
    {
        Serial.println("Hej fra Arduino'en ^_^");
        delay(2000);
    }
}
```

MegunoLink

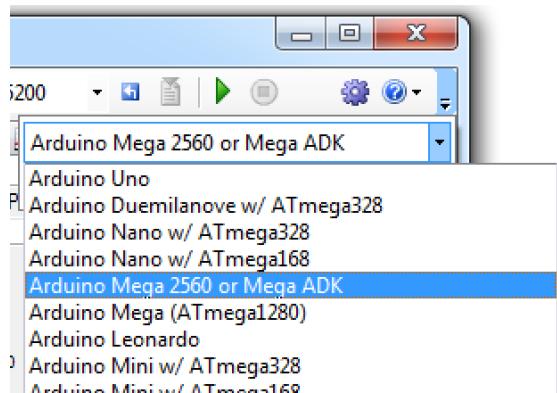
Installér MegunoLink. Inden MegunoLink kan anvendes til at programmere Arduino'en, skal det opsættes til Arduino Mega2560. Det skal angives, hvor AVRdude.exe og avrdude.conf er placeret. Disse er begge vedlagt dokumentations-CD'en. Kopiér dem derfor over på computeren til en passende placering. Tryk herefter på opsætningsikonet vist på figur 3.68 og angiv deres placering.



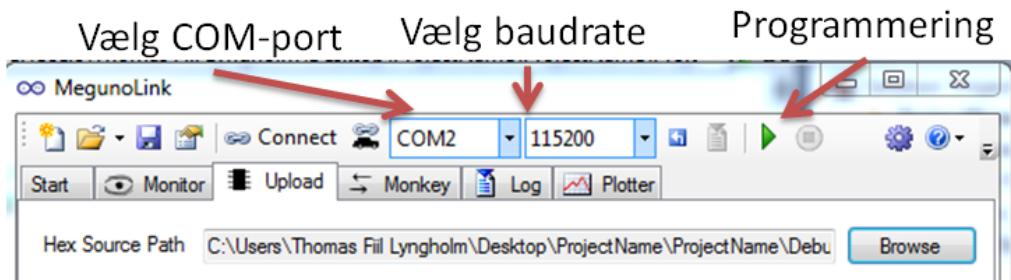
Figur 3.68.



Figur 3.69. Konfigurationsmuligheder for MegunoLink. Afslut med OK.

*Figur 3.70.*

På fig 3.70 vælges 'Arduino Mega 2560' som den enhed man ønsker at programmere. Under fanen "Upload" browser man nu frem til den ønskede *.hex fil fra AtmelStudio. Denne ligger i projektets mappe under "/Debug". Herefter tilsluttes Arduino'en, og det observeres hvilken COM-port den tildeles. Dette vælges i MegunoLink sammen med baudrate (oftest 115200). For at programmere klik på den grønne pil.

*Figur 3.71.*

Tryk herefter på "Connect" og vælg fanen "Monitor". Det er nu muligt at monitorere det serielle output fra Serial (UART0) fra Arduino Mega2560.

Import af projekt i Atmel Studio

For at åbne projektet i Atmel Studio kopieres softwaren for dronen fra den relative sti: "..\CD\Bilag\Software\DroneSoftware\" over på PC'en. Projektet kan herefter åbnes fra mappen Main_drone. I det vedlagte projekt er alle afhængigheder sat op.

3.7.1.2 AeroQuad Flight Control Board v. 2

Til opsætning og test af AeroQuad Flight Control Board og AeroQuad drone er PC applikationen AeroQuad Configurator anvendt. Programmet giver mulighed for at kompilere og uploadere bootloader og firmware med brugerens ønskede indstillinger, samt kalibrere og

monitorere dronens sensorer. Dette afsnit vil beskrive, hvordan AeroQuad Configurator er anvendt i dette projekt. Beskrivelsen er opdelt på følgende vis:

- 1 Installation af AeroQuad Configurator v. 3.2
- 2 Kommunikation med AeroQuad 32 Flight Control Board v. 2
- 3 Upload af AeroQuad 32 Flight Software
- 4 Konfiguration af dronestellet
- 5 Pre-Flight Checkout

Installation af AeroQuad Configurator v. 3.2

Installationen af AeroQuad Configurator applikationen er ligetil, og kan gøres til Windows platforme enten fra CD'en:

"...\\Deployment\\Main Controller\\AeroQuadConfigurator_v3.2Win\\setup.exe"

Eller fra AeroQuad's Google repository:

<https://code.google.com/p/aeroquad/downloads/list>

På Google repository'et er det desuden muligt at installere en OSX version af Configurator'en. Det er væsentligt at bemærke, at den nyeste Flight Software kun er anvendelig med den nyeste Configurator software. Til dette projekt er anvendt flight software v. 3.2 og Configurator v. 3.2. Da Configurator'en anvender en online compiler, kompileres den nyeste version af Flight Softwaren altid, og det er derfor en god idé, at være i besiddelse af den nyeste version af Configurator'en.

Kommunikation med AeroQuad 32 Flight Control Board v. 2

For at kunne kommunikere med AeroQuad32 board'et, er der tre trin, der skal udføres én gang ved første anvendelse af dronen:

- 1 Installation af virtual COM port
- 2 Opsætning af board i DFUSe mode
- 3 Installation af USB bootloader driver

Installation af virtuel COM port

Installér den virtuelle COM port driver fra CD'en: "...\\Deployment\\Main Controller \\AeroQuad32Setup\\01_OneTimeSetup\\01_InstallVirtualComPortDriver\\VCP_V1.3.1_Setup.exe". På CD'en forefindes både en 32 og 64 bit version.

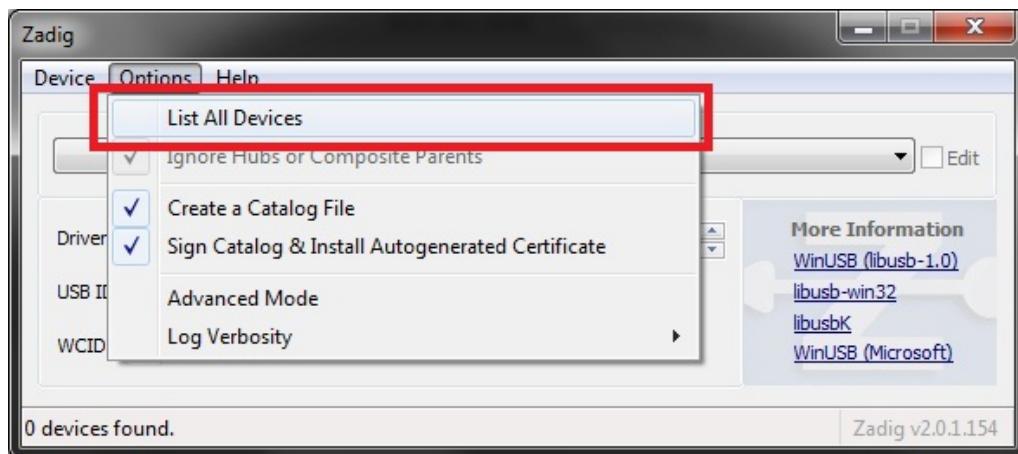
Efter endt installation tilsluttes AeroQuad32 board'et en USB port, og driverinstallationen færdiggøres automatisk. Windows 7 brugere skal være opmærksomme på, at kommunikation med board'et via USB 3.0 ikke fungerer. Dette er ikke noget problem for Windows 8 brugere. Board'et skal være tilsluttet under resten af AeroQuad-opsætningen.

Opsætning af board i DFUSe mode

For at muliggøre upload af ny firmware til board'et, skal board'et sættes i DFUSe mode. For at gøre dette højreklikkes på batch filen "...\\Deployment\\Main Controller\\AeroQuad32Setup\\01_OneTimeSetup\\02_ConfigureBoardIntoDFUEMode\\reset.bat", og "Rediger" vælges. Den første linje i filen ændres til den tildelte COM port fra VCP(virtual COM port) installationen. Pga. begrænsninger i batch filen, anbefales det, at COM porten tildeles en af portene 0-9. Den tildelte COM port kan ses og ændres i Windows enhedshåndtering.

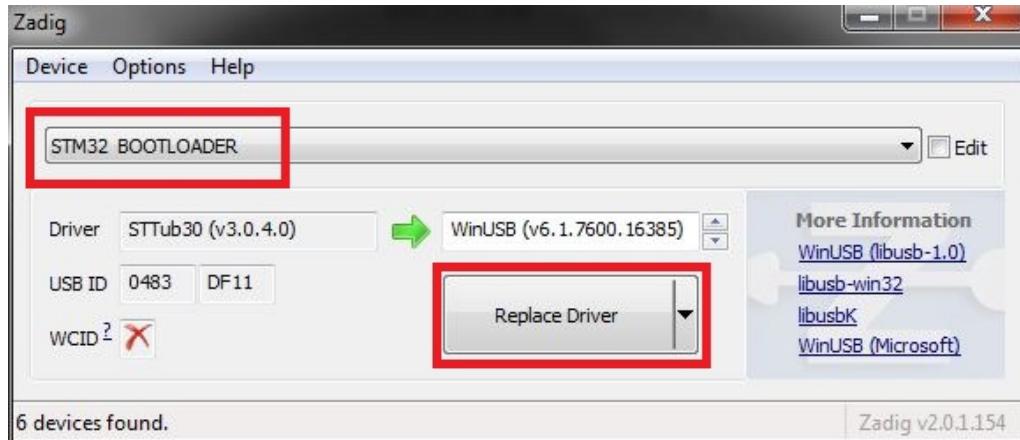
Installation af USB bootloader driver

For at kunne kommunikere med AQ32 board'et, er det sidste trin at placere en bootloader på board'et, der kan tage sig af kommunikationen med PC'en. Dette gøres ved at åbne programmet på CD'en "...\\Deployment\\Main Controller\\AeroQuad32Setup\\01_OneTimeSetup\\03_InstallUSBDriver\\zadig.exe". Vælg "Options"→"List All Devices".



Figur 3.72.

Vælg "STM32 BOOTLOADER" på drop down menuen og tryk "Replace Driver". Hvis "STM32 BOOTLOADER" ikke er på listen, er board'et ikke i DFUSe mode. Prøv at åbne reset.bat igen.

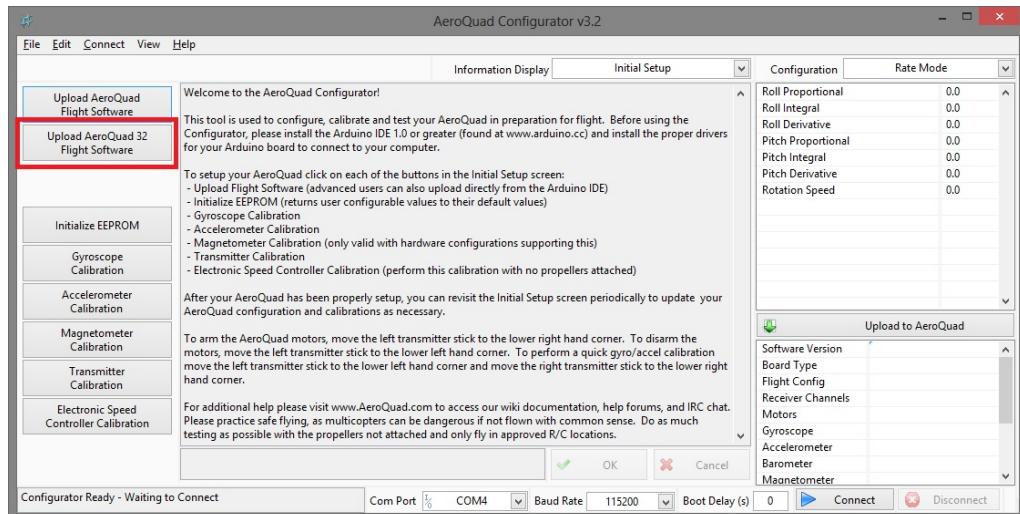


Figur 3.73.

Board'et er nu klar til opsætning og upload af ny firmware vha. AeroQuad Configurator.

Upload af AeroQuad 32 Flight Software

I dette afsnit uploads den korrekte firmware til AeroQuad board'et vha. AeroQuad Configurator. Programmet åbnes og på startskærmen trykkes på "Upload AeroQuad 32 Flight Software".



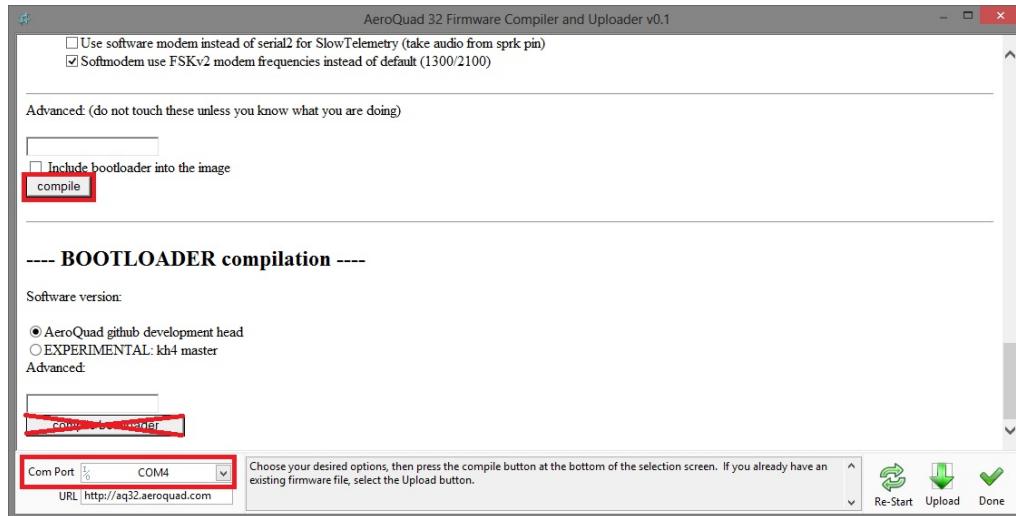
Figur 3.74.

Programmet tilgår herefter en online compiler, hvor der kan vælges forskellige opsætninger af board'et. Følgende skal vinges af ud over standardopsætningen:

- 1 "Quad+" under "Flight Configuration"
- 2 "Magnetic heading hold (uses mag instead of gyro)" under "Sensors"
- 3 "Barometric altitude hold" under "Sensors"
- 4 "Battery Monitor" under "Battery Monitor"

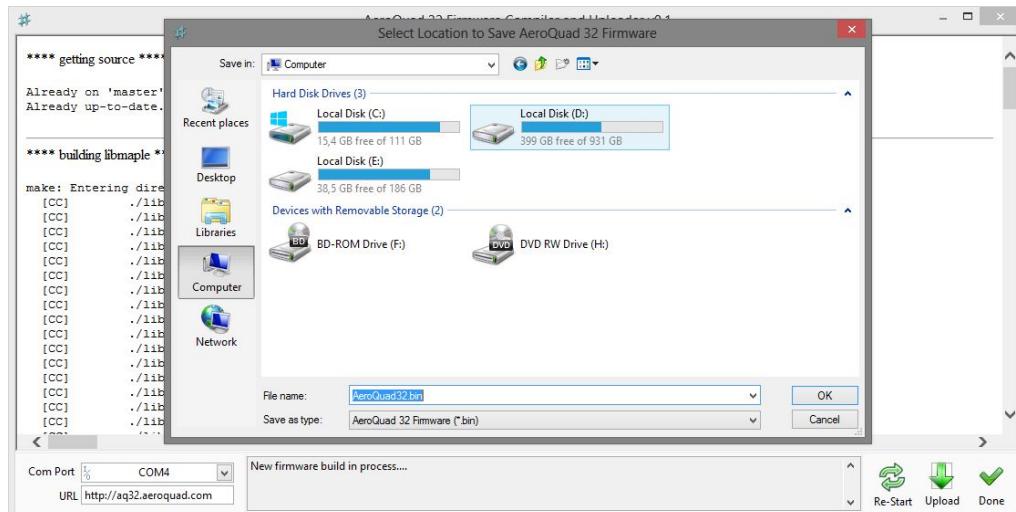
5 "Use SERIAL3 instead of USB (for xbee/bluetooth etc)" under "Telemetry"

Herefter vælges den anvendte COM port, og der trykkes på "Compile".

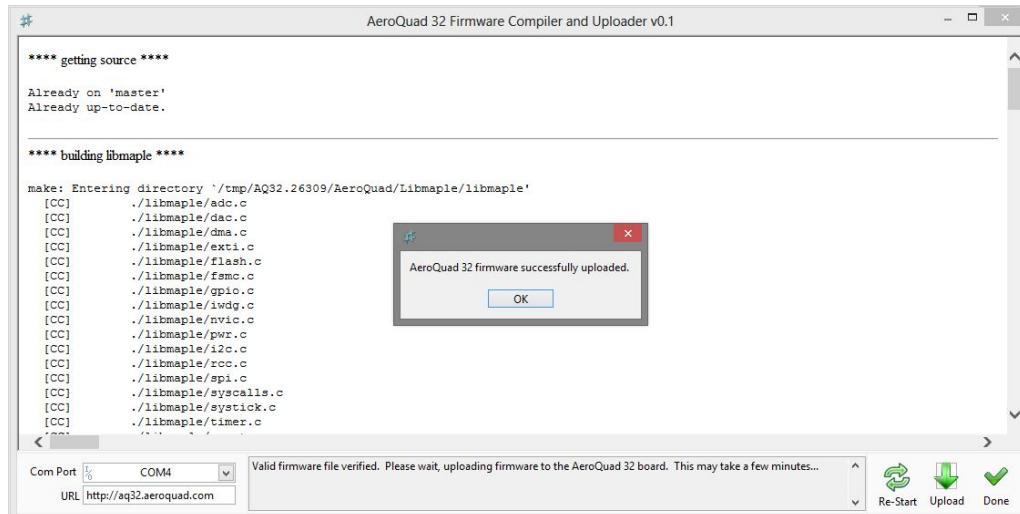


Figur 3.75.

Den kompilerede bin fil gemmes på PC'en, hvorefter den uploades til AeroQuad board'et.



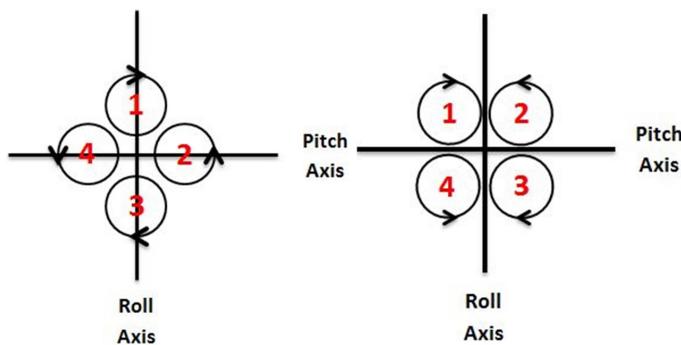
Figur 3.76.

**Figur 3.77.**

Med indstillingen "Use SERIAL3 instead of USB (for xbee/bluetooth etc)" aktiveret, skal fremtidig kommunikation mellem AeroQuad board og AeroQuad Configurator foregå via et USB til RS232 kabel koblet til AeroQuad board'ets "SER" port. Board'ets USB port skal dog stadig anvendes, hvis der skal uploades ny firmware eller bootloader.

Konfiguration af dronestellet

På figur 2.5 i afsnit 2.6 ses et diagram over hardwareopsætningen af systemet. Dronen kan i Configurator'en sættes op til at flyve i "+ mode" eller "X mode". De to forskellige modes er afbildet på figur 3.78, og beskriver hvordan motorerne er nummereret ift. dronens stel. I dette projekt er det besluttet at flyve dronen i "+ mode".

**Figur 3.78.** Tv. "+ mode". Th. "X mode".

Pre-Flight Checkout

Inden flyvning er det en god idé at gennemgå punkterne i nedenstående liste, for at sikre korrekt opsætning af dronen. Det anbefales at udføre checkout-listen uden vinger monteret.

- 1 Initialiser EEPROM
- 2 Kalibrer gyroskop
- 3 Kalibrer magnetometer
- 4 Kalibrer transmitter
- 5 Kalibrer ESC'er
- 6 Kontroller korrekt tilslutning af ESC'er til AQ32 jf. den valgte konfiguration af dronestellet ("+ mode" eller "X mode"). Kontroller ligeledes, at motorerne roterer den rigtige vej, og at de rigtige vinger er placeret på de rigtige motorer.

3.7.1.3 Android Applikation

I dette afsnit vil opsætning af udviklingsmiljø til Android blive gennemgået. Først skal følgende elementer hentes:

- Android Development Tool¹⁵
- Java Runtime Environment ¹⁶

Download en passende version og installer begge ¹⁷.

Når begge er installeret, startes Eclipse op. Vælg her et passende workspace. Til at start med installeres opdateringer. Her anvendes Android SDK Manager. Vælg "Windows->"Android SDK Manager". Herefter installeres nødvendige opdateringer til Android. Følgende opdateringer skal hentes:

- API 16+19.
- Android Support Library
- Google Play Service
- Google USB Driver
- Android SDK Tools
- Android SDK Platform-tools

Efter alle opdateringer er installeret, importeres koden. Kopier indholdet fra mappen App der ligger på den vedlagte CD, ind i Android workspace. Der er tale om tre mapper:

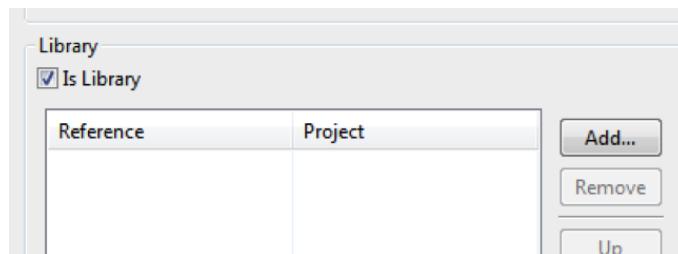
- "appcompat_v7_3"
- "google-play-services_lib"
- "DroneControl"

Vælg "File"->"Import"->"General"->"Existing Projects into Workspace". Vælg de tre mapper, der blev kopieret i forrige trin. Tryk "Finish". Tjek eventuelt at "appcompat_v7_3" og "google-play-services_lib" er sat op til at være biblioteker ved at højreklikke på mappen og vælg "Properties". Vælg "Android" i sidebjælken. Tjek at "Is Library" er afkrydset.

¹⁵<http://developer.android.com/sdk/index.html#download>

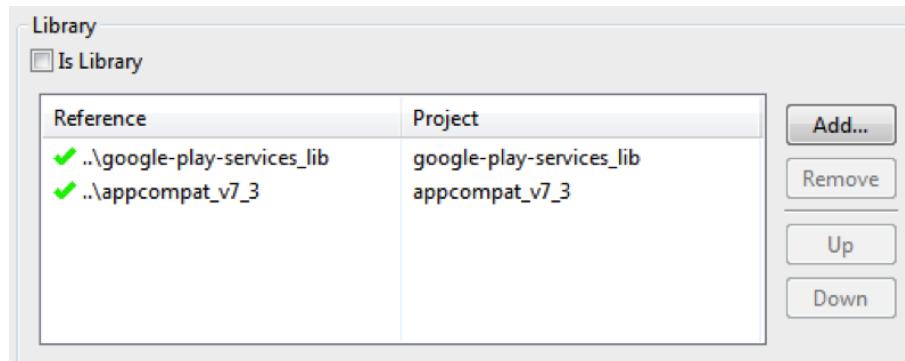
¹⁶<http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html>

¹⁷Installationsguide til ADT: <http://developer.android.com/sdk/installing/index.html?pkg=adt>



Figur 3.79. viser "properties" for bibliotek.

For "DroneControl" skal der samme sted være referencer til de to biblioteksprojekter.



Figur 3.80. viser at projektet har referencer til bibliotekerne.

Anvend "Remove" og "Add" hvis dette ikke er tilfældet.

I tilfælde af fejlmeldelse om manglende Google Maps API key, anvendes følgende online-guide¹⁸.

DroneControl projektet er nu klar til videre arbejde.

3.7.1.4 Server

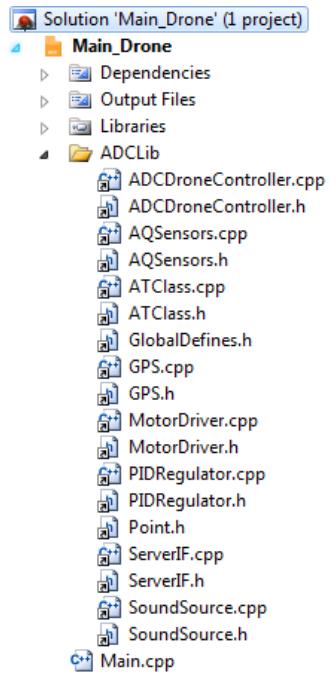
For at kunne tilgå serveren fra internettet, i forbindelse med resten af ADC-systemet, kører serveren på en virtuel Windows Server maskine, leveret af IHA's IT afdeling. På den virtuelle maskine er der åbnet for internettet på port 5050, der er den port serveren benytter. Dette muliggør tilgang til serveren fra internettet. Serveren tilgås via url'en: "<http://iha-11471.ihah.dk:5050>", når den kører på den virtuelle maskine. Skal serveren køre på en anden computer, skal der være åbnet for internetadgang på port 5050, og firewall'en skal acceptere tcp forbindelser på denne port. Serverprojektet benytter .NET Framework 4.5 og skal åbnes i Microsoft Visual Studio 2012 eller nyere. Så snart projektet bygges i Visual Studio, kopieres de nødvendige mapper og filer til outputmappen¹⁹.

¹⁸https://developers.google.com/maps/documentation/android/start#obtain_a_google_maps_api_key

¹⁹Alt efter om build configuration er "Debug" eller "Release", er outputmappen henholdsvis "/ADCServer/bin/Debug" eller "/ADCServer/bin/Release" set fra projektmappens rodmappe.

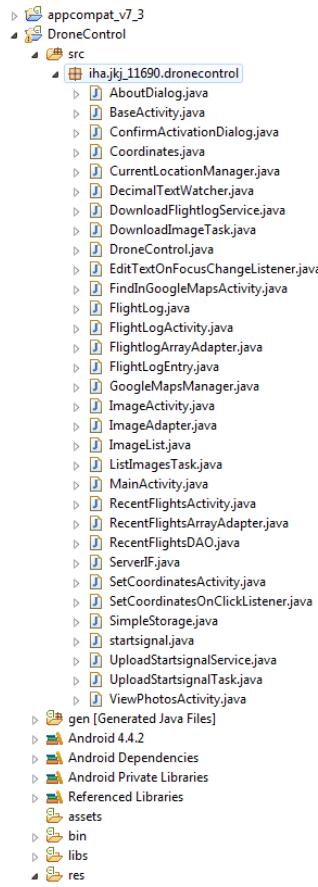
3.7.1.5 Source-kodeoversigt

Drone

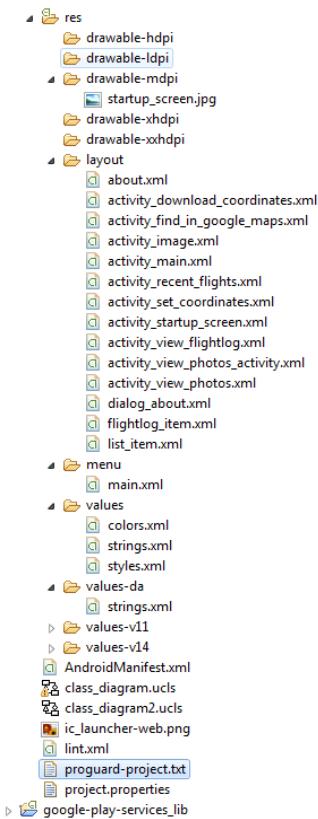


Figur 3.81. Filoversikt for drone.

Smartphone Applikation

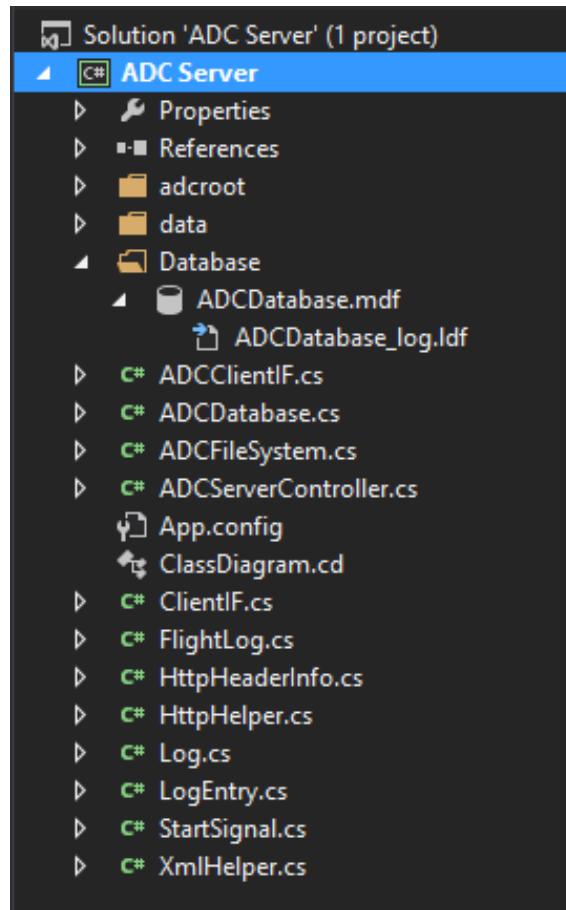


Figur 3.82. Filoversigt for smartphone applikationen(1/2).



Figur 3.83. Filoversigt for smartphone applikationen(2/2).

Server



Figur 3.84. Filoversigt for server.

3.7.2 Yderligere softwareelementer og -klasser

I dette afsnit beskrives softwareelementer og -klasser, der ikke er fremgår direkte af de tidlige designfaser, men som alligevel har væsentlig betydning for systemet.

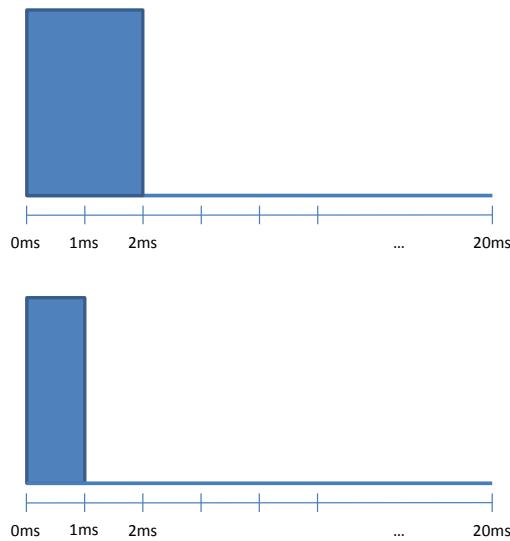
3.7.2.1 Motor driver

For at kunne styre dronen med Arduino'en, er det nødvendigt at vide, hvordan en transmitter/receiver styrer dronen, da Arduino'en essentielt skal overtage dennes rolle. Når dronen styres med en transmitter/receiver, forbinder receiver'en til Aeroquad (AQ) board'ets fire styrepins. Styrepinsene kontrollerer hastigheden af dronens tre rotationsakser: Pitch, yaw og roll, og motorhastigheden: Throttle²⁰. Det er signalet fra receiver'en til disse pins, Arduino'en skal imitere.

²⁰Se afsnit 3.7.2.5 "PID-regulering" for uddybning af rotationsakser og motorhastighed.

RC PWM

Signalet fra receiver'en følger PWM RC-standarden²¹, hvor selve styringen sker med et PWM-signal med en on-tid på 1-2ms og en periodetid på 20ms og dermed en frekvens på 50Hz, som det ses på figur 3.85:



Figur 3.85. RC PWM-protokol.

Denne type signal sendes til hver af de fire styre-pins på AQ board'et. On-tiden på 1-2ms omsættes til en værdi, der bestemmer hastigheden. En on-tid på 1ms svarer til den laveste hastighed, og en on-tid på 2ms svarer til den højeste hastighed.

Opsætning af timerne

For at efterligne dette signal på Arduino'en, benyttes dennes 16-bit timer-registre. Timerne benyttes i "fast PWM mode" og "clear on compare match"²², hvor timerne tælleregistre nulstilles, hver gang de når en top-værdi. Ved at benytte en 16-bit timer, kan en høj top-værdi opnås. Dette vil betyde en høj opløsning på duty cycle'en, hvilket vil medføre en høj opløsning for selve styringen. Udregningen for output PWM-frekvensen af timerne ses her, jf. databladet for Arduino'ens micro processor, ATmega2560²²:

$$f_{OCnxPWM} = \frac{f_{clk_I/O}}{N \cdot (1 + TOP)} \quad (3.1)$$

Hvor $f_{OCnxPWM}$ er frekvensen på timerens tilhørende outputpin, $f_{clk_I/O}$ er ATmega2560's clock frekvens, N er værdien af timerens prescaler og TOP er den værdi, der tælles op til i

²¹http://en.wikipedia.org/wiki/Servo_control

²²Se databladet for ATmega2560 på CD'en under: "...\\Bilag\\Datasheets\\ATmega2560 (doc2549).pdf"

tælleregisteret, før timeren tikker over. Prescaler'en er en fast skaleringsfaktor, der benyttes til at skalere processorens clock. PWM-frekvensen afhænger dermed af TOP-værdien, når prescaler'en er defineret. ATmega2560's clock-frekvens er 16MHz. Ved at sætte den ønskede frekvens $f_{OCnxPWM}$ til 50Hz, og en prescaler på 8, kan ligningen løses for at finde den TOP-værdi, der giver det ønskede 50Hz PWM-signal:

$$50Hz = \frac{16000000MHz}{8 \cdot (1 + TOP)} \quad TOP = \frac{16000000MHz}{50Hz \cdot 8} - 1 \quad TOP = 39999 \quad (3.2)$$

Ved at benytte en prescaler på 8, opnås den højest mulige oplosning for tælleregisteret. De mulige prescaler-værdier på ATmega2560 er: 1, 8, 64, 256 og 1024. Med en prescaler på 1 fås den TOP-værdi, der medfører en PWM-frekvens på 50Hz, til 399999. Da der kun er 16 bit i tælleregisteret, er det ikke muligt at opnå en PWM-frekvens på 50Hz med denne prescaler værdi. Benyttes en prescaler på 64, fås TOP værdien til 3999. Altså vil en prescaler på 8, medføre den højeste valide TOP-værdi, og dermed give den højeste oplosning for duty cyclen. Opsætningen af timerne, der medfører "fast PWM mode", "clear on compare match", prescaling på 8, og en PWM-frekvens på 50Hz, ses her:

```
TCCRnA = _SET(COMnA1) | _SET(COMnB1) | _SET(COMnC1) | _SET(WGMn1);
TCCRnB = _SET(CSn1) | _SET(WGMn2) | _SET(WGMn3);
ICRn = 39999;
```

Her er "n" tallet for den timer der sættes op, ICRn sætter den pågældende timers TOP-værdi og `_SET()` er en makro, der sætter en bit:

```
#define _SET(bit) (1 << (bit))
```

De anvendte timere er ATmega2560's timer 3 og timer 4, der begge er 16-bit timere, og konfigureres ens. Et 50Hz PWM-signal med en periode på 20ms og en on-tid på henholdsvis et og to millisekunder, svarer til en duty cycle på henholdsvis fem og ti procent:

$$T_{PWM} = \frac{1}{50Hz} = 20ms \quad (3.3)$$

$$duty\ cycle_{1ms} = \frac{1ms}{20ms} = 5\% \quad (3.4)$$

$$duty\ cycle_{2ms} = \frac{2ms}{20ms} = 10\% \quad (3.5)$$

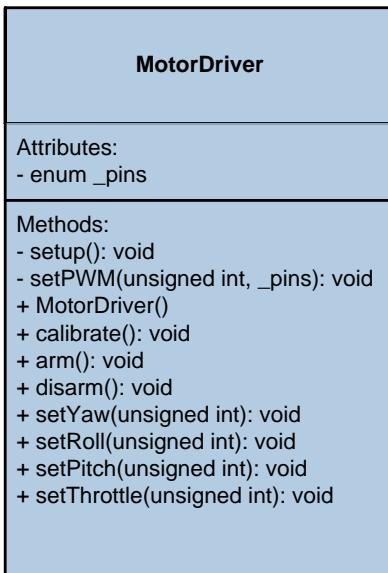
Værdierne for tælleregistrene, der vil producere et PWM-signal med en on-tid på henholdsvis et og to millisekunder, er:

$$OCRnA = 5\% \cdot 39999 = 1999 \quad (3.6)$$

$$OCRnA = 10\% \cdot 39999 = 3999 \quad (3.7)$$

Hvor OCRnA er tælleregistret for den pågældende timer. Værdierne sættes til 2000 og 4000, da dette er mere overskueligt i koden, og ikke vil have nogen betydelig indvirkning

på on-tiden. Det er dermed værdier mellem 2000 og 4000 for tælleregisteret, der benyttes til at generere PWM-signalen, der sendes til AQ board'et fra Arduino'en. Motor driveren er pakket ind i en klasse, som det ses på figur 3.86.



Figur 3.86. MotorDriver-klassen.

Klassen indeholder metoder til at sætte de tre rotationsakser: Yaw, pitch og roll og motorhastigheden: Throttle, hvor den private metode setPWM benyttes til at styre duty cyclen på styre-pins'ene. *setup*-metoden opsætter timerne, mens *arm* og *disarm* henholdsvis armerer og desarmerer dronen. *calibrate*-metoden løber et program igennem, der sætter yaw, pitch, roll og throttle i yderpositioner, og som benyttes når yaw, pitch, roll og throttle skal kalibreres. Slutelig indeholder enum'en *_pins* navnet på de forskellige styre-pins der kontrolleres på AQ board'et, og benyttes til at switche på hvilken pin, der skal skrives til forskellige steder i koden.

3.7.2.2 Arduino kode til C++

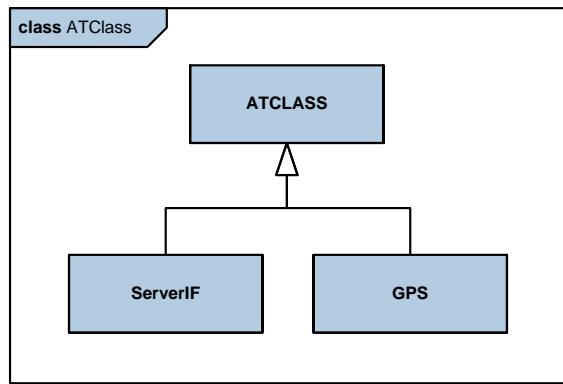
Det anvendte 3G+GPS shield er købt hos Cooking-Hacks, der er en online elektronik webshop for hobbyingeniører. På deres hjemmeside²³ findes tutorials og kodeeksempler til de fleste af deres produkter herunder det anvendte shield. I dette projekt er koden til kommunikation med shield'et stærkt inspireret herfra.

Kodeeksemplerne, der ligger på deres hjemmeside, er Arduino sketches, og eftersom dette projekt i fremtiden ikke ønskes begrænset til Arduino, er sketch'ene porteret til C++ kode.

Til at kommunikere med shield'et anvendes AT-kommandoer. For at have et optimalt kodegenbrug er der lavet en basisklasse indeholdende metoder til at kommunikere via AT. Fordelen ved at designe en basisklasse er, at alle klasser der laves på baggrund af denne klasse, arver dens metoder. Derved behøves der ikke en reel instans af klassen. De

²³<http://www.cooking-hacks.com/documentation/tutorials/arduino-3g-gprs-gsm-gps>

klasser der arver fra denne AT-klasse, er *GPS* og *ServerIF*. En fremtidig implementering af kamera og SD kort på dronen ville ligeledes skulle arve fra AT-klassen. På figur 3.87 ses et klassediagram for klassernes arvehierarki.



Figur 3.87. Diagram over arv for *GPS* og *ServerIF*.

GPS-klassen står for al kommunikation med GPS'en. Der er lavet metoder til at hente GPS-informationer, opsætning af GPS, deserialisering af modtaget GPS-information samt andre hjælpemetoder.

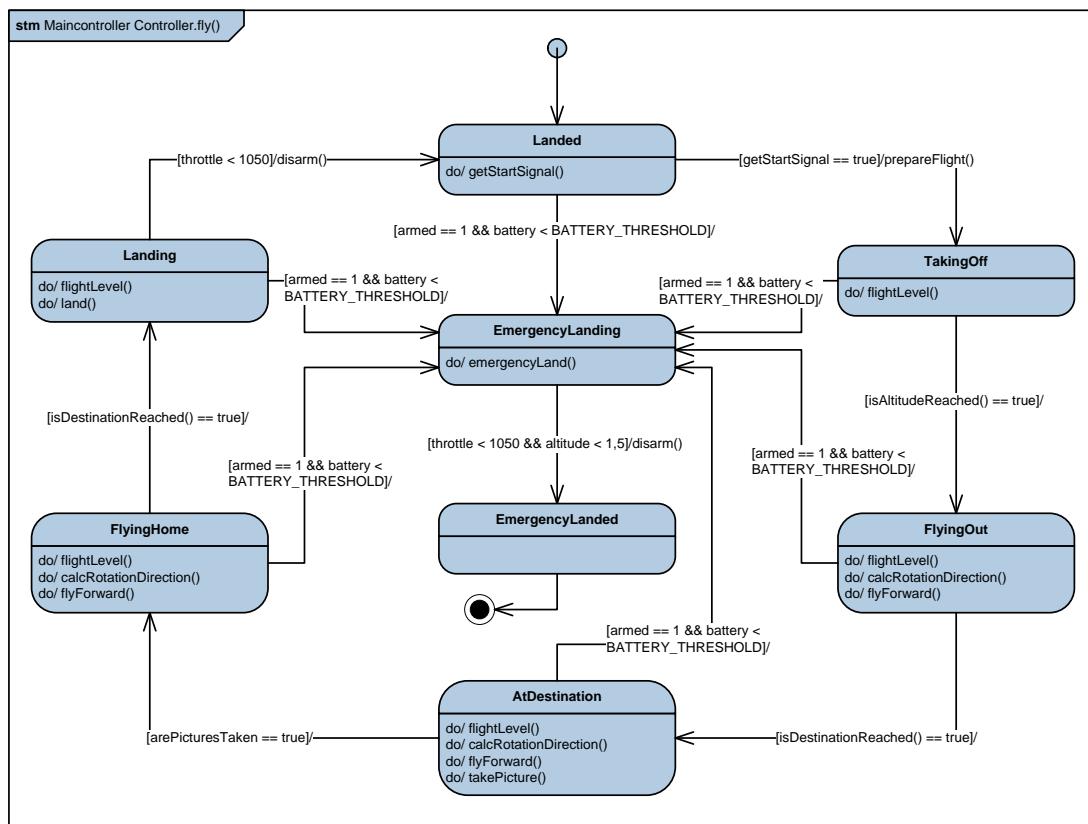
ServerIF-klassen er ansvarlig for al interaktion med ADC server fra dronens side. Dvs. den står for at hente startsignal samt afsende logposter. Al funktionalitet mht. serialisering og deserialisering af HTTP-beskeder ligger ligeledes i klassen.

3.7.2.3 ADCDroneController state machine

Dronens *ADCDroneController*-klasse indeholder alle de metoder, der benyttes i dronens main. En af klassens primære metoder er

```
void fly()
```

Metoden står for at læse dronens sensorer (GPS og AQ sensorer), opdatere PID-regulatorerne og til sidst styre dronen ud fra regulatorernes output og dronens tilstand. Metoden er designet som en state machine med otte states, der definerer forskellig adfærd for dronen afhængig af den aktuelle state. Figur 3.88 viser tilstandsdiagrammet for *fly*-metoden.



Figur 3.88. Tilstandsdiagram for *Controller*-klassens *fly*-metode.

I det følgende bliver de otte states forklaret.

Landed

Ved første kald af *fly*-metoden, er dronen i tilstanden "Landed". I denne state er dronen desarmeret, og forsøger at få et startsignal fra serveren. Når dronen har fået et startsignal, opdateres tilstanden til "TakingOff".

TakingOff

I denne tilstand letter dronen ved at regulere flyvehøjden mod den ønskede flyvehøjde. Når flyvehøjden er nået, opdateres tilstanden til "FlyingOut".

FlyingOut

I denne tilstand flyver dronen mod destinationen. Dronen regulerer stadig flyvehøjden, og så længe dronen er inden for den ønskede flyvehøjde, beregnes og reguleres der desuden på flyveretningen. Når højde og flyveretning er inden for de definerede marginer, reguleres der på flyvehastigheden mod destinationen. Når destinationen er nået, opdateres tilstanden til "AtDestination".

AtDestination

I denne tilstand har dronen nået sin destination, og forsøger at holde sig på destinationen ved at udføre samme opgaver som i "FlyingOut-tilstanden. Derudover tager dronen billeder. Når billederne er taget, opdateres tilstanden til "FlyingHome".

FlyingHome

I denne tilstand er dronen på vej hjem fra sin destination. Udoer destinationen, er dronens adfærd den samme som i "FlyingOut-tilstanden. Når dronen er nået hjem, opdateres tilstanden til "Landing".

Landing

I denne tilstand har dronen nået sin hjemdestination, og udfører landing. Når dronen er landet og har desarmeret sine motorer, opdateres tilstanden til "Landed".

EmergencyLanding

Dronen har til enhver tid i de seks tidlige tilstænde mulighed for at gå til denne tilstand, hvis batteriniveauet falder til under minimumsværdien. I denne tilstand foretages en hurtigere landing end den i "Landing-tilstanden. Når dronen er landet og har desarmeret sine motorer, opdateres tilstanden til "EmergencyLanded".

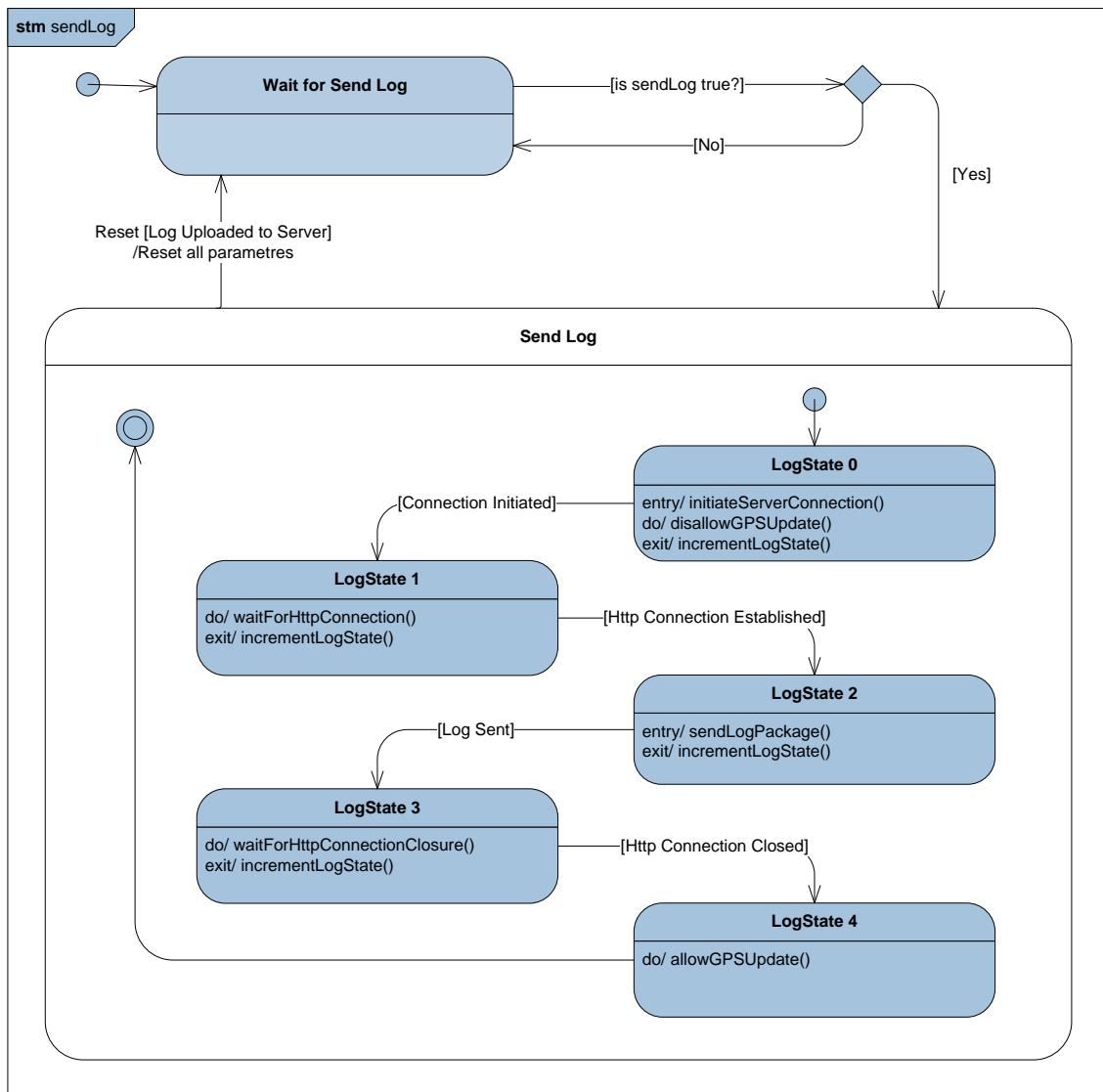
EmergencyLanded

Dronen kommer i denne tilstand, hvis dronen har udført en nødlanding på baggrund af lavt batteriniveau. Denne tilstand kan kun forlades, ved at genstarte Arduino'en. Dette vil ske naturligt, hvis batteriet skiftes.

3.7.2.4 Log state machine

Ved at implementere en interrupt-styret state machine, er det muligt at udføre regulering af flyvekurs og -højde samtidig med afsendelse af logposter. Dog skal det bemærkes, at det ikke er muligt at modtage GPS-informationer samtidig med afsendelse af log. Dette skyldes, at GPS- og 3G-modem begge sidder på shield'et, og derfor ikke kan anvendes på samme tid. Logposterne sendes hvert 30. sekund, og tager ikke mere end 3-4 sekunder at sende. Derfor er GPS-dødtiden ikke betydelig for systemet, da dronens position ikke ændrer sig væsentligt i dette tidsinterval.

På figur 3.89 ses et tilstandsdiagram for implementering af logfunktionaliteten.



Figur 3.89. Tilstandsdiagram for afsendelse af log.

Der er to overordnede states, hvor state to har fem substates. De to overordnede states afspejler, hvorvidt der sendes en log, eller om der blot ventes på at skulle sende en. For at imødekomme en ikke blokerende funktionalitet er afsendelse af logposter inddelt i substates. Hver substate består af en simpel funktion, der startes, hvorefter det igen er muligt at foretage regulering. Dette er muligt da shield'et har sin egen processor. Ved at anvende AT-kommandoer styres de HTTP-specifikke dele på shield'et uden at blokere resten af systemet. Til at styre ventetiden mellem de forskellige substates anvendes et interrupt. Herved opnås en ikke blokerende logafsendelse.

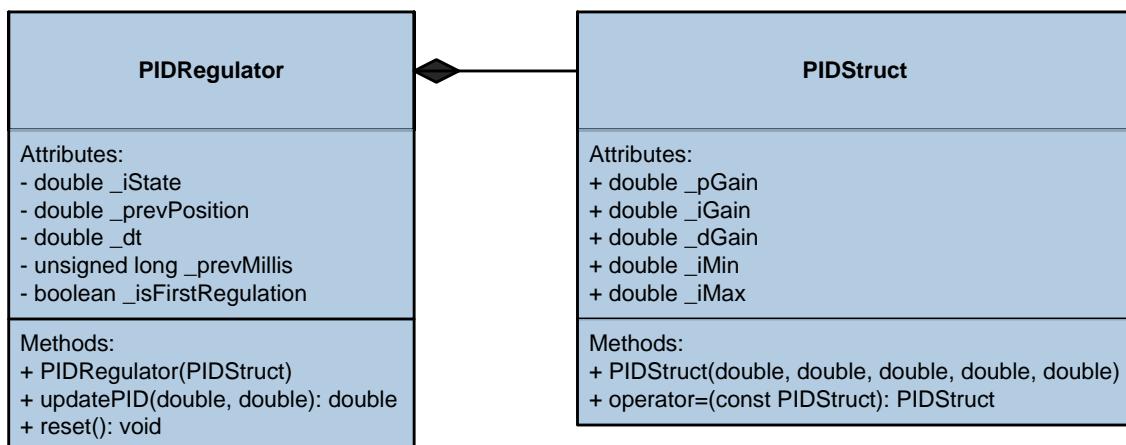
3.7.2.5 PID-regulering

Til at styre den autonome flyvning er der implementeret fire PID-regulatorer. De fire regulatorer har forskellige PID-parametre, sætpunkter, inputs og formål, og de vil derfor

blive beskrevet hver for sig i det følgende afsnit. Til alle reguleringerne er den samme PID-klasse anvendt, og denne vil først blive beskrevet.

PID-klasse

Klassen *PIDRegulator* står for at styre motorernes hastighed ud fra de målte sensorværdier. På figur 3.90 ses klassediagrammet for regulatoren.



Figur 3.90. Klassediagram for PIDRegulator.

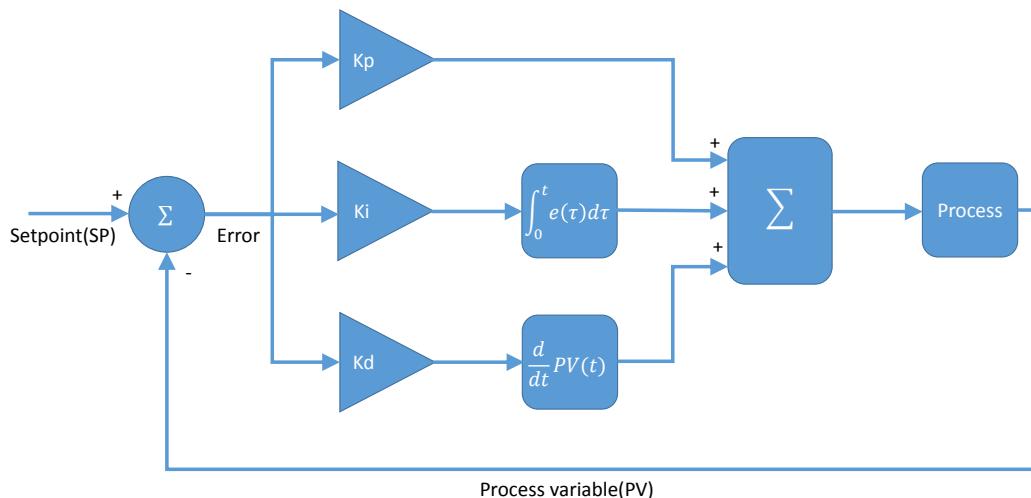
Klassens primære funktionalitet ligger i metoden

```
double updatePID(double error, double position)
```

Metoden tager fejlen og positionen for den aktuelle regulator som parametre, og returnerer et output, der senere bruges som hastighedsparameter i en af *ADCDroneController*-klassens flyvefunktioner. Returværdien er summen af P-, I- og D-leddene. P-leddet er blot en faktor af den øjeblikkelige fejl. I-leddet er en integration af alle tidligere fejl, og er derfor afhængig af tidsforskellen fra sidste regulering/beregning. Ligeledes er D-leddet afhængig af tiden, da leddet beregnes som den afledte position.

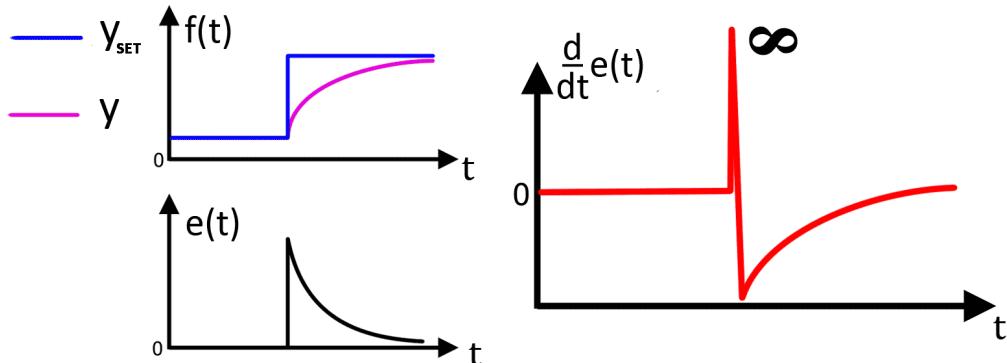
Regulering af throttle

Throttle-reguleringen skal regulere dronens throttle og dermed dronens højde. Når dronen tændes, nulstilles AQ32 board'ets barometer, og dets nuværende barometriske tryk sættes lig et referencetryk på 0. Dronens flyvehøjde er altså relativ ift. dette opstartspunkt. Procesvariablen er givet ved det barometriske tryk målt af AQ32 board'ets barometer, og giver dronens højde målt i meter over referencepunktet. Trækkes feedback'et fra sætpunktet, fås den fejl, der føres ind i regulatorens P- og I-led. D-leddet kan implementeres på to forskellige måder: Error feedback eller rate feedback. På figur 3.91 ses implementeringen af error feedback, og som det fremgår af figuren, er fejlen ind i D-leddet den samme som for P- og I-leddene.



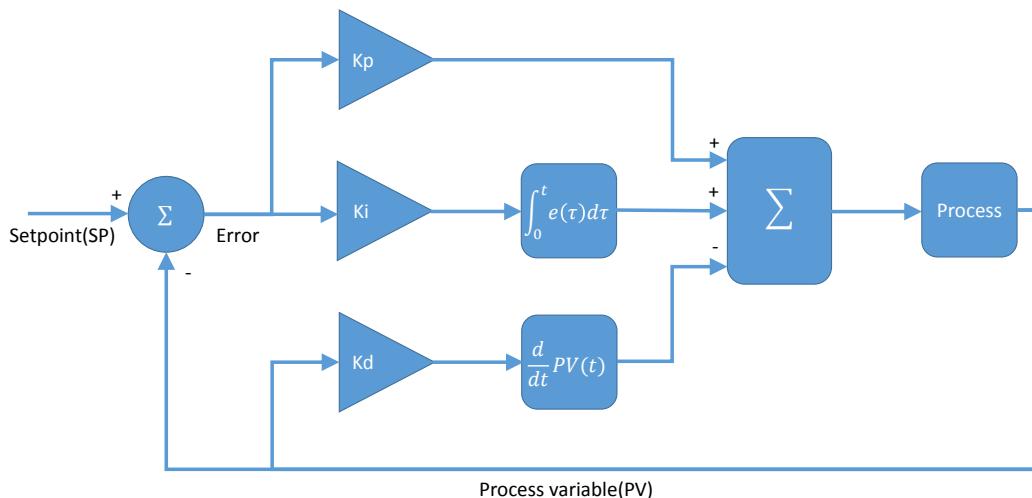
Figur 3.91. Error feedback.

Ulempen ved denne implementeringsform er et fænomen kaldet "derivative kick". Derivative kick opstår, når sætpunktet pludseligt ændres betydeligt, da en ændring i sætpunktet betyder en direkte ændring i fejlen. D-leddet beregner hældningen af fejlen, og vil derfor ved en øjeblikkelig ændring i sætpunktet give et potentieligt uendeligt stort kick til processen. Dette er illustreret på figur 3.92.



Figur 3.92. Derivative kick.

Dette kan heldigvis let undgås ved blot at benytte PV som input til D-leddet, som vist på figur 3.93. Denne implementeringsform kaldes rate feedback. Ændringen i PV er den samme som ændringen i fejlen, bortset fra når sætpunktet ændres. En ændring i sætpunktet påvirker ikke PV, og der forekommer derfor ikke derivative kicks. Derudover er fortegnet for $\frac{d}{dt}e(t)$ og $\frac{d}{dt}PV(t)$ forskellige, og D-leddets output skal derfor trækkes fra P- og I-leddenes output. Alle PID regulatorer i dette projekt er implementeret med rate feedback.

**Figur 3.93.** Rate feedback.

Da barometeret er meget følsomt over for vibrationer, sollys og direkte luftstrømninger, er barometeret pakket ind i skum og sammen med resten af sensorerne monteret på et stødabsorberende underlag, der isolerer sensorerne fra de vibrerende motorer.

Regulering af yaw

Yaw-reguleringen bestemmer dronens kurs ved at rotere dronen om z-aksen. PV til yaw-regulatoren beregnes af AQ board'et ud fra magnetometer- og accelerometermålinger. Fejlen beregnes som et gradtal fra 0 til 180, da den aktuelle kurs aldrig kan være mere end 180 grader fra den ønskede kurs. Yaw-regulatoren bruges kun under ud- og hjemflyvning.

Regulering af pitch

Pitch-reguleringen bestemmer dronens fart i fremadretningen ved at rotere dronen om y-aksen. PV til pitch-regulatoren er afstanden til destinationen i meter. Udover at styre dronens fart, skulle pitch-regulatoren have været brugt under op- og nedstigning, samt under fotografering på destinationen for, sammen med roll-regulatoren, at holde dronen geografisk fastlåst. Det sidste er ikke implementeret.

Regulering af roll

Roll-reguleringen roterer dronen om dens x-akse. PV til roll-regulatoren er afstanden til destinationen i meter. Roll-regulatoren er ikke implementeret, men skulle have været brugt under op- og nedstigning, samt under fotografering på destination for, sammen med pitch-regulatoren, at holde dronen geografisk fastlåst.

3.7.2.6 Geografiske beregninger

Beregning af kurs

Main controller-klassen *ADCDroneController* har *calcHeading*-metoden med prototypen:

```
double calcHeading(Point currentGPS, Point dstGPS);
```

Metoden tager to GPS-koordinatsæt som parametre, og beregner den geografiske kurs i grader imellem de to punkter. Formlen, der anvendes til udregningen, ser ud som følger²⁴:

$$\theta = \text{atan}2(\sin(\Delta\lambda) \cdot \cos(\phi_2), \cos(\phi_1) \cdot \sin(\phi_2) - \sin(\phi_1) \cdot \cos(\phi_2) \cdot \cos(\Delta\lambda)) \quad (3.8)$$

Hvor ϕ og λ er hhv. længde- og breddegrad i radianer. Denne formel bruges løbende til at beregne den aktuelle kurs, og der er derfor ikke behov for at tage hensyn til indledende og afsluttende kurs på en flyvning.

Beregning af afstand

Main controller-klassen *ADCDroneController* har *calcDist*-metoden med prototypen:

```
double calcDist(Point currentGPS, Point dstGPS);
```

Funktionen tager to GPS-koordinatsæt som parametre, og beregner den geografiske afstand i meter imellem de to punkter. Formlen, der anvendes til udregningen, kaldes Haversine, og beregner afstanden mellem de to punkter i en cirkelbane langs Jordens gennemsnitlige radius. Formlen ser ud som følger²⁴:

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right) \quad (3.9)$$

$$c = 2 \cdot \text{atan}2(\sqrt{a}, \sqrt{1-a}) \quad (3.10)$$

$$d = R \cdot c \quad (3.11)$$

Hvor ϕ og λ er hhv. længde- og breddegrad i radianer, R er Jordens gennemsnitlige radius(6.378.137m²⁵) og d er afstanden imellem de to punkter.

²⁴<http://www.movable-type.co.uk/scripts/latlong.html>

²⁵http://en.wikipedia.org/wiki/Earth_radius