MDD ロボットチャレンジ 2005

モデル審査用資料(2005/10/10)

チーム FSEL(エフセル)
立命館大学情報理工学部

代表者：丸山勝久，山本哲男

筏谷浩樹，石塚聖啓，原田彬直，平井孝

1. はじめに

本資料は, MDD ロボットチャレンジ 2005 におけるモデル審査用に作成したものである.

２. 開発の流れと作成したモデル

今回の開発の手順を（１）～(4)で説明する．作成したモデルに関しては，以下のように分類し，巻末に添付する.

(a) 要求モデル(1/2 ～ 2/2)
(b) 分析モデル(1/10 ～10/10)
(c) 設計モデル(1/15 ～ 15/15)
(d) 変換(1/5 ～ 5/5)
(e) 詳細設計モデル(1/13 ～ 13/13)
(f) フレームワーク化（1/)
(g) 最終版クラス図(1/1)

(1) 要求の分析

MDD ロボットチャレンジ 2004(pp.95-98)「チャレンジャへの指令」に基づき，要求モデルを作成した．図１に，与えられたシステム全体の概要図を示す.

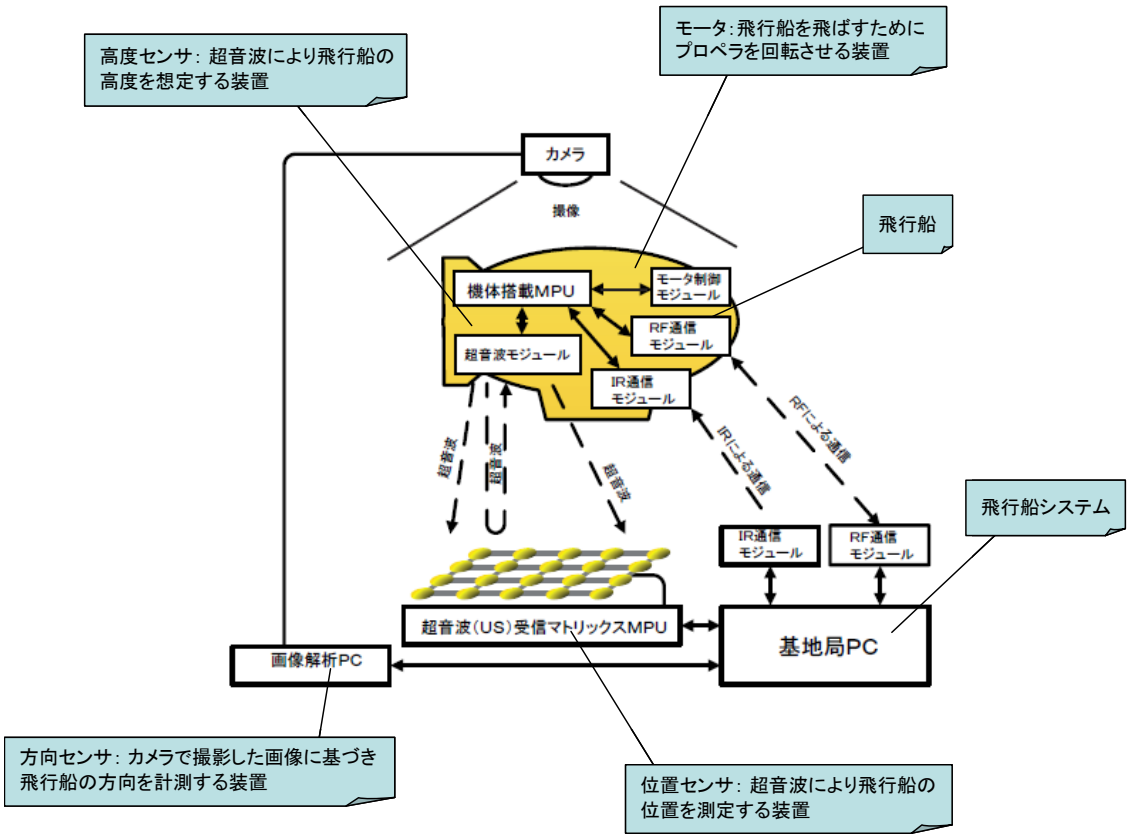


図１： システム概要図

Haluna-1 を用いて，野生動物観測を実施する調査システムのユースケース図を要求モデル(1/2)に示す．今回は，指令により，飛行船システムのプロトタイプを作成する．作成する飛行船システムのユースケース図を要求モデル(2/2)に示す．

われわれのチームはハードウェアおよびドライバを作成せず，飛行船を制御システム(基地局 PC)のプログラムだけを開発（納品）対象とする．このため，直前での試験がしやすいように，競技者だけでなく，保守者から見た機能も検討した．システム概念図に基づき作成した飛行船システムの概念モデルを分析モデル(1/10)に示す．システム化の対象は，競技者と保守者を除くクラスである．

表１，表２に非正常系を考慮した際の検討項目と，検討結果を示す．今回のシステムでは，位置情報の取得できない場合，および，原因不明で飛行船が目的地（中間地点）に到着しない場合に対処することとした．

表１：非正常系の検討

現象		障害	検出	対策
飛行船が一定時間以内に目的地に到着しない	飛行船の制御不能	基地局から指令送れず	通信エラーが連続して発生 飛行船のタイマ(今回は実装しない)	システムは停止 飛行船は着陸
		モータ回らず	目視	とらない
	センサ情報が間違い	高度情報が得られない	通信エラーが連続して発生 高度データが連続して想定範囲外	地上USの位置情報からの計算結果を利用するように切り替える
		位置情報が得られない	通信エラーが連続して発生 位置データが連続して想定範囲外 (USゾーン内)	システムは停止 飛行船は着陸
		方向情報が得られない	通信エラーが連続して発生 方向データが連続して想定範囲外	地上USの位置情報の差分から推測するように切り替える
	原因不明 複合要因	—	タイマにより検出	目的地を切り替える

表２：非正常系の検討（結果）

現象		障害	検出	対策
飛行船が一定時間以内に目的地に到着しない	飛行船の制御不能	基地局から指令送れず	対処しない	
		モータ回らず		
	センサ情報が間違い	高度情報が得られない	対処しない	
		位置情報が得られない	妥当な位置情報が取得できない	旋回飛行を実行し，制御可能になるまで待つ
		方向情報が得られない	対処しない	
	原因不明 複合要因	—	タイマにより検出	目的地を切り替える

要求モデル(2/2)の各ユースケースに対して，アクティビティ図を記述することで，要求分析を行った．今回の開発では，要求者（発注者）と要求定義者が同一のため，ユースケース図においてシナリオを記述せず，アクティビティ図により利用手順を明確化した．機能を表現したユースケースは，以下の(a)～(f)である．

(a) 飛行経路を与える：分析モデル(2/10)
(b) 飛行船を自動操縦で飛ばす：分析モデル(3/10)
(c) 飛行特性を与える：分析モデル(4/10)
(d) US センサの位置を与える：分析モデル(5/10)
(e) モータとセンサの試験を行う：分析モデル(6/10)
(f) モータの出力値を決定する：分析モデル(7/10)
(g) 飛行船の状況を表示する：分析モデル(9/10)

分析モデル(8/10)，分析モデル(10/10)はアクティビティを詳細化したものである．また，モータ，高度センサ，位置センサ，方向センサ，タイマのユースケースに関しては，機能はほぼ明確でなるという理由からアクティビティ図を記述しなかった．

(2) 設計モデルの作成

PIM モデルとして，クラス図，シーケンス図，状態図を作成した．クラス図ではシステムの静的構造を，シーケンス図と状態図では動的側面を表現する．分析モデル(1/10)のクラス図に基づき，主要なクラスを抽出し，シーケンス図と状態図を作成する過程でクラス図を洗練した．

プラットフォーム非依存レベルにおいて，最終的に作成したクラス図を設計モデル(1/15)に示す．飛行船システムのインタフェース(クラス図における飛行船 UI クラスの各メソッド)に関するシーケンス図を設計モデル(2/15)～設計モデル(12/15)に示す．

(a) US センサの位置を設定する：設計モデル(2/15)
(b) 飛行特性を設定する：設計モデル(3/15)
(c) 飛行経路を設定する：設計モデル(4/15)
(d) 自動飛行を開始する：設計モデル(5/15)
(e) テスト飛行を開始する：設計モデル(6/15)

設計モデル(7/15)～設計モデル(12/15)は，上記シーケンス図の一部を詳細化したものである．状態図を設計モデル(13/15)～設計モデル(15/15)に示す．飛行船の制御アルゴリズムを飛行モードにより変更できるように状態を管理する．また，自動飛行とテスト飛行に必要な情報の入力を検査するために，これを状態で管理する．タイマは飛行予測時間の経過を検査するために導入し，時間が経過したかどうかで状態を切り替えるようにした．

(3) 詳細設計

PIM モデルから PSM モデルに変換する．変換は主に以下に示す３点で行った．動作環境

は，UNIX (FreeBSD および Linux)，開発言語は C++とした．

(a) 通信に関するデータのスレッドかとデータのロック

高度情報，位置情報，方向情報を取得するオブジェクトをメインスレッドと別のスレッドで実行するように，高度センサクラス，位置センサクラス，方向センサクラスに対して，変換(1/5)に示す変換操作を適用した．受信器<U>が外部からデータを受け取る部分で，UNIX の pthread による生成されたスレッドで独立に動作する．また，送受信において共有されるデータのロックには， Read-Write Lock パターンを用い，SimpleLock クラスの提供する lock()メソッドおよび unlock メソッドで実現する．データ書き込みおよび読み込み前に lock()メソッドを呼び出し，データの書き込みおよび読み込み後に unlock()を呼び出す．今回の開発では，比較的単純なロック器を作成したが，SimpleLock クラスを置き換えることで簡単に複雑なロック器を用いることができる．

変換を適用した後のクラス図の一部を変換(2/5)に示す．また，受信に関するシーケンスを詳細設計(13/13)に示す．送信に関しては，別スレッド化する必要性はないと判断し，同期呼び出しで実現した．

(b) クラスから属性への変換

集約関係にある<<entity>>タイプクラスを属性に変換する．用意した変換は，順序つき集約関係<<ordered>>，識別子による検索つき集約<<mapped>>である．ぞれぞれの変換の様子を変換(3/5)と変換(4/5)に示す．変換後の<<entity>>タイプクラスは，<<data>>オブジェクトクラスとした．

(c) クラス名と操作名の英語化

C++で実装することを考え，クラス名と操作名をアルファベット表記に変換した．変換表を変換(5/5)に示す．フィールド名と引数名に関しては，もともとルファベット表記であったため，変換表は不要である．

上記(a)～(c)の変換を適用した後のクラスを，詳細設計(1/13)に，シーケンス図を詳細設計(2/13)～詳細設計(13/13)に示す．詳細設計(2/13)～詳細設計(12/13)は，設計(2/15)～設計(12/15)にそれぞれ対応する．詳細設計の際には，ループの終了条件と分岐の条件の詳細化も手動で同時に行った．

(4) フレームワーク化

(3)で作成した詳細設計に関して可変可能部分を検討し，フレームワーク化を行った．導入する可変部分をフレームワーク化(1/1)に示す．

(5) 飛行アルゴリズムの決定

飛行特性として，次の４つのマップをそれぞれのモードに対して用意し．モータの出力を決定する．

(a) 高度差マップ：Characteristics クラスの upperMap

U = (高度差[m]を正規化したもの，上昇モータの出力値)

(b) 角度マップ - 左モータ用：Characteristics クラスの leftMap

L = (目的地と進行方向の角度を正規化したもの，左モータの出力値)

(c) 角度マップ - 右モータ用：Characteristics クラスの rightMap

R = (目的地と進行方向の角度を正規化したもの，右モータの出力値)

(d) 距離マップ：Characteristics クラスの distMap

W = (現在地と目的地との距離を正規化したもの，左右モータの出力への重み)

モータの出力値 = (U, L × W, R ×W)

(6) 最終版クラス図の作成

最終的に完成した詳細設計モデルを最終版クラス図(1/1)に示す．このクラス図におけるドメインを図２に示す．

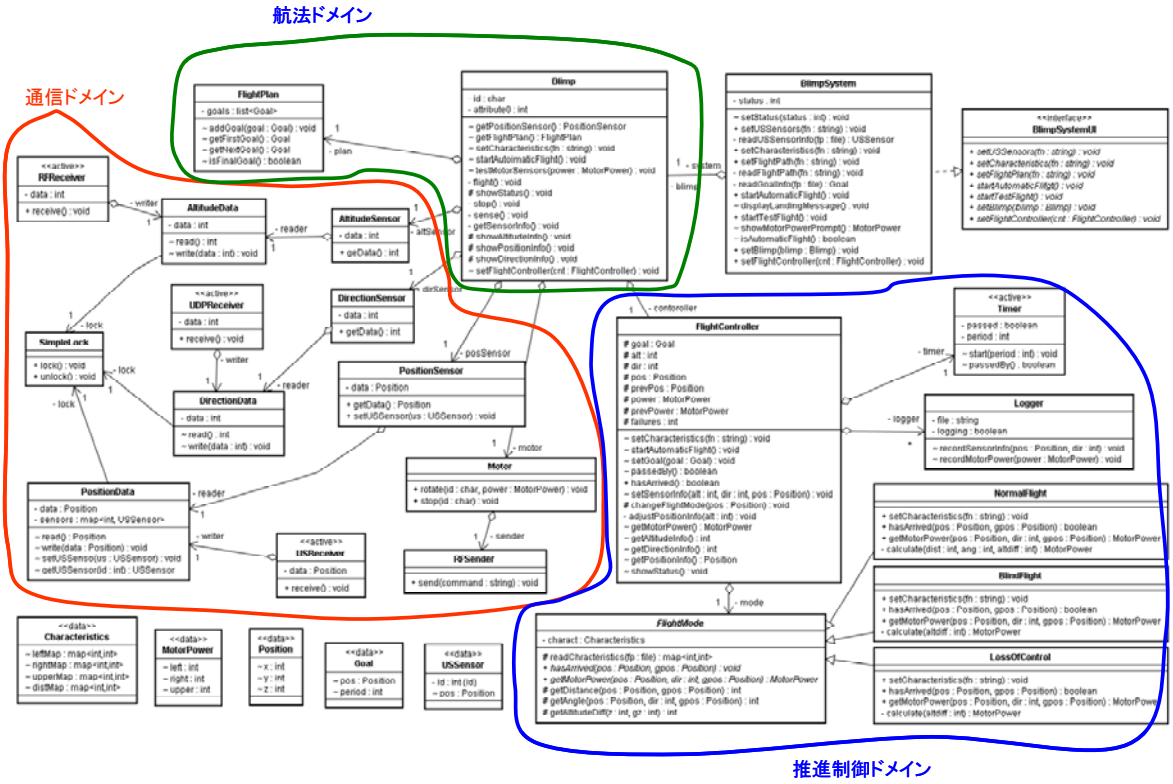


図２：最終版クラス図におけるドメイン

(7) 実装

最終版クラス図の実装においては，main 関数（あるいは main から最初に呼ばれる関数）

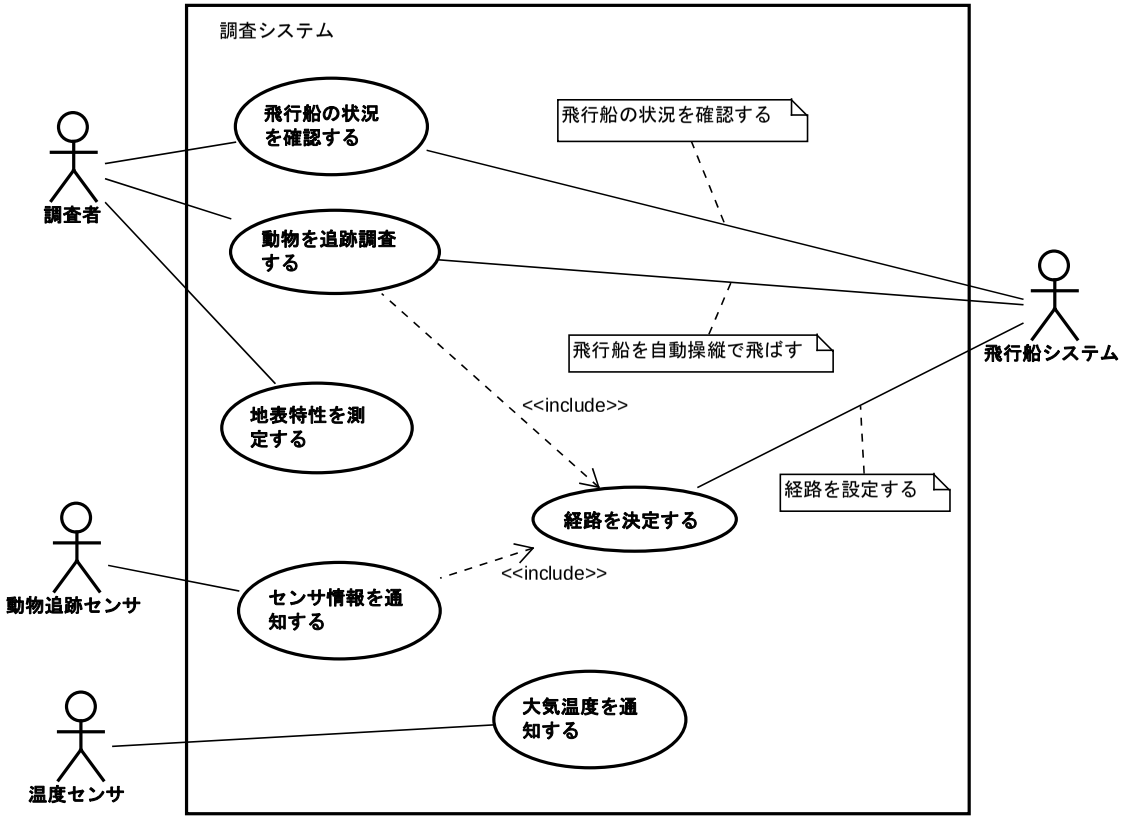
で３つの受信器用のスレッドを生成し，それをそれぞれのオブジェクトに割り付ける．その後，BlimpSystemUI クラスのメソッドを呼び出すことで，ユースケースに基づく操作を実現する．次の 2 つの実装を作成中である．

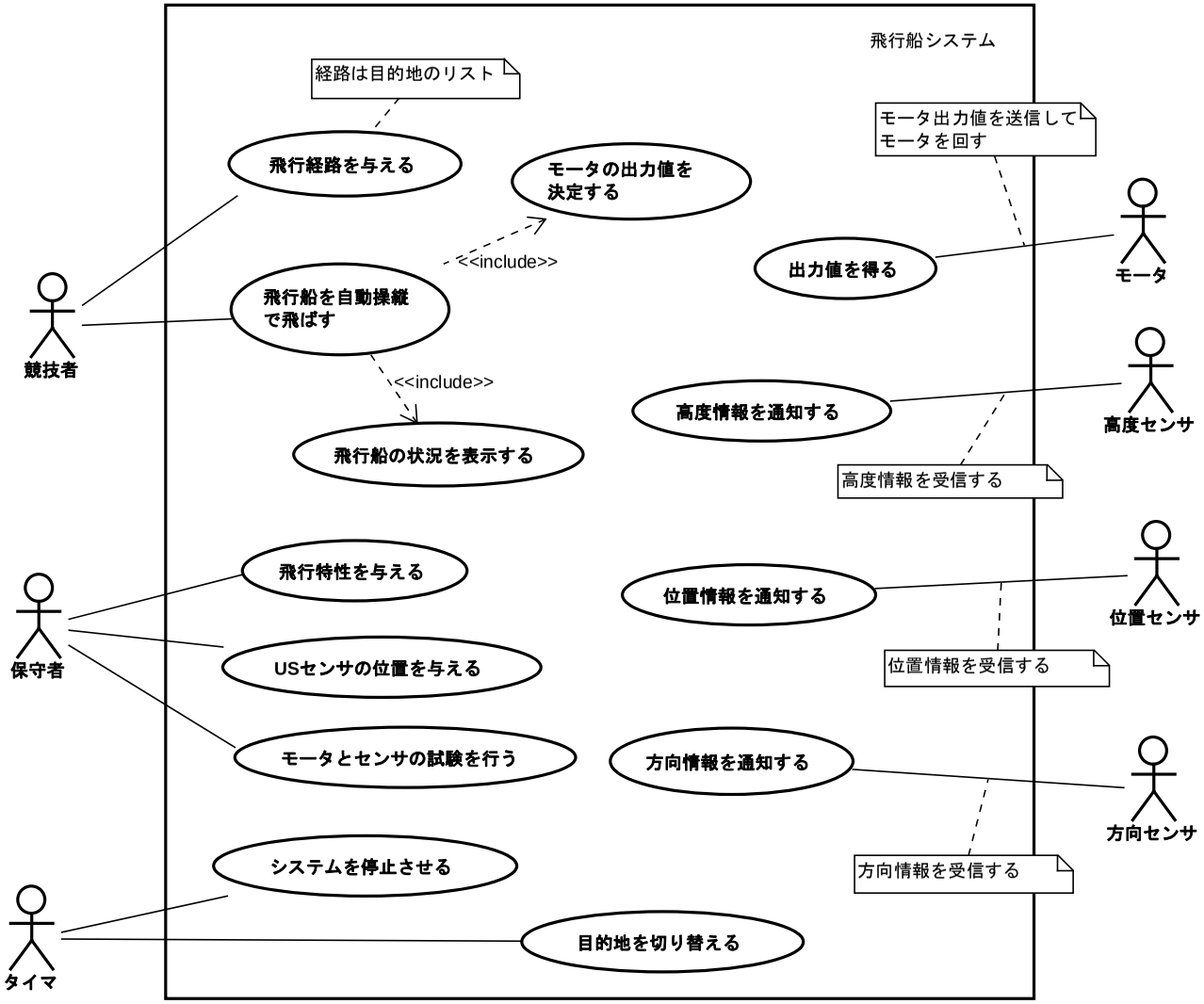
fsel：自動飛行用プログラム（本番で使用）

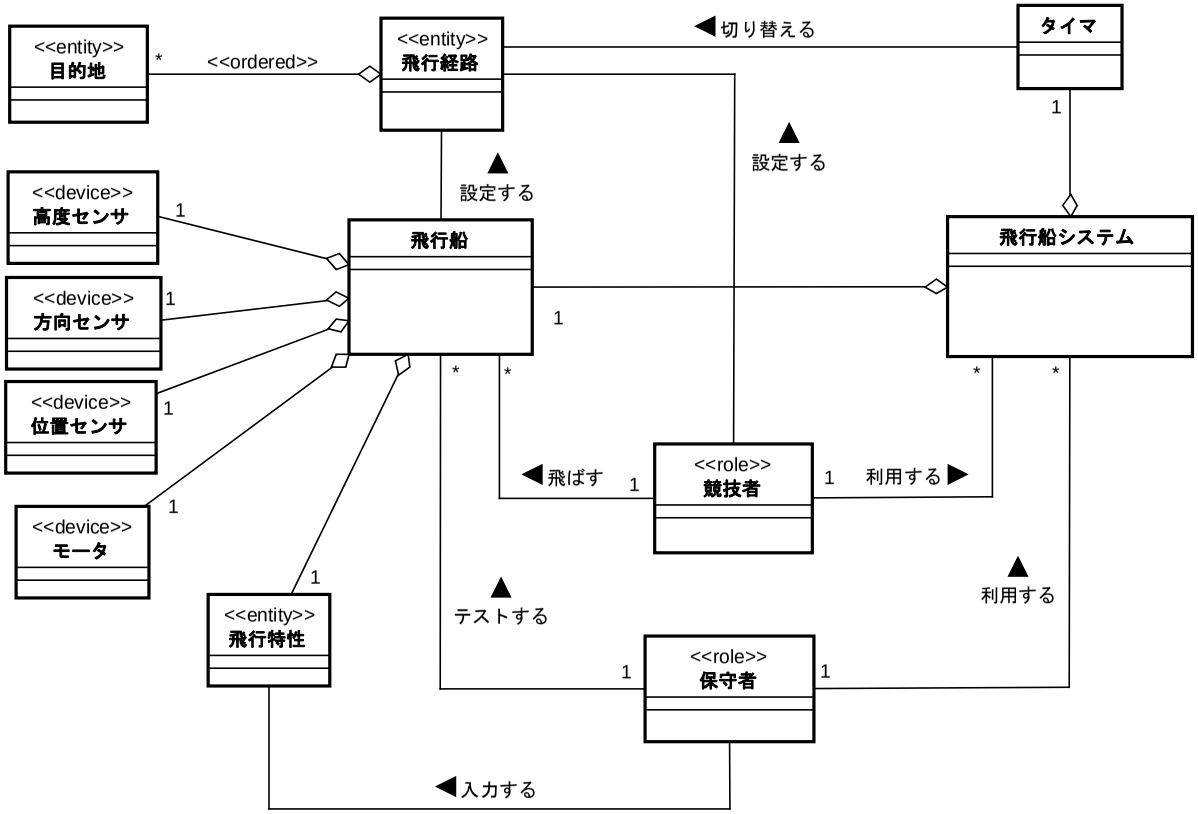
fsel-manual：テスト飛行用プログラム

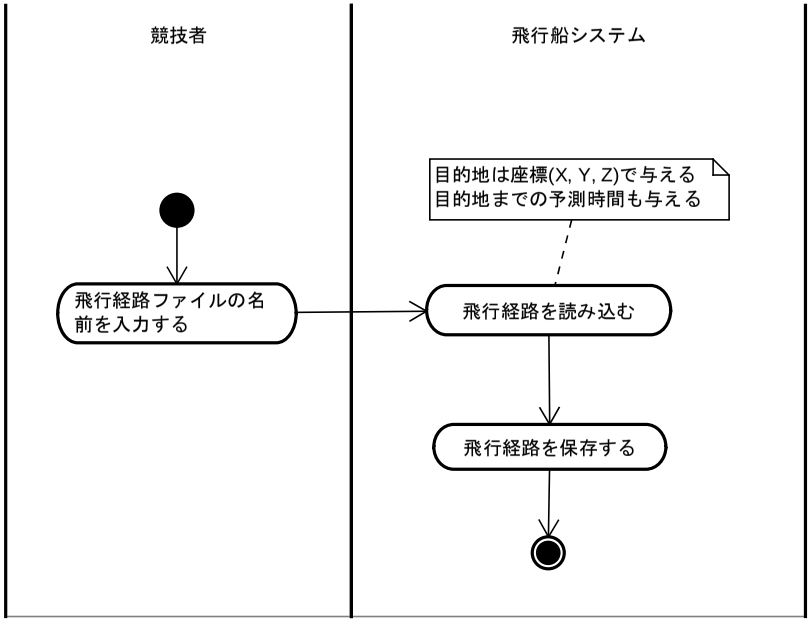
3. おわりに

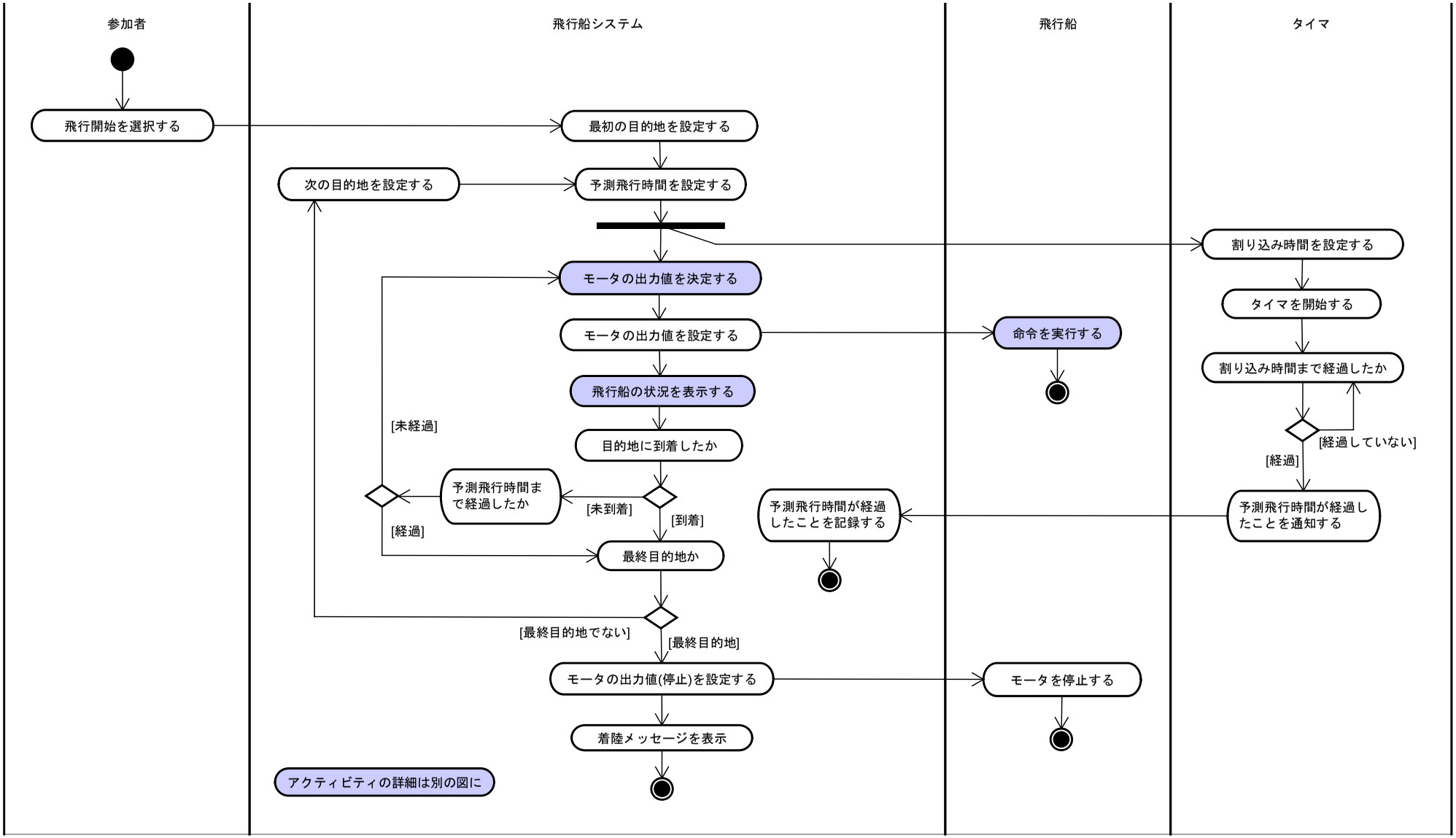
本企画に参加するにあたり，ご尽力ご教授いただきました主催者の皆様，また，提出したモデルをご審査いただく審査員の皆様に感謝いたします．

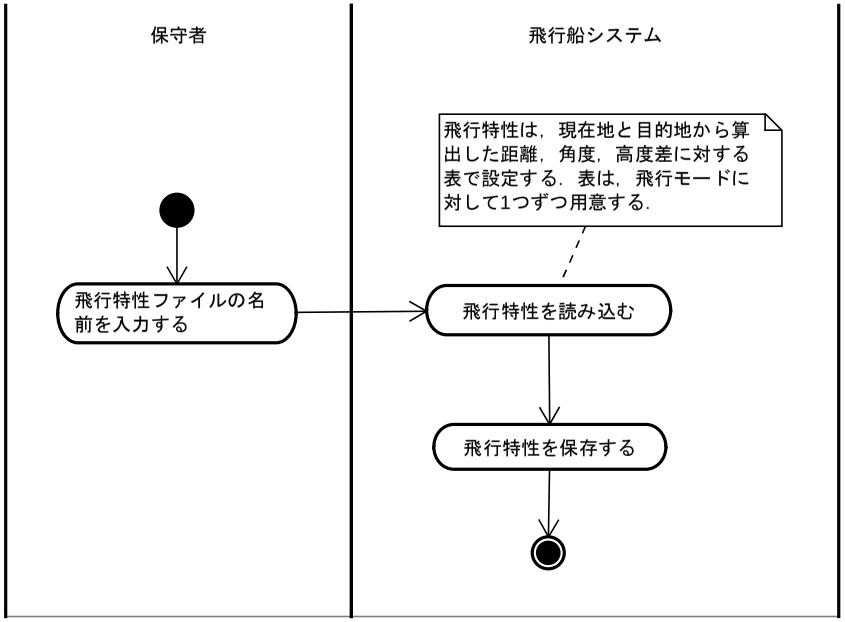

調査システム
飛行船の状況を確認する
動物を追跡調査する
地表特性を測定する
経路を決定する
センサ情報を通知する
大気温度を通知する
飛行船の状況を確認する
飛行船を自動操縦で飛ばす
経路を設定する
<<include>>
<<include>>
調査者
動物追跡センサ
温度センサ
飛行船システム

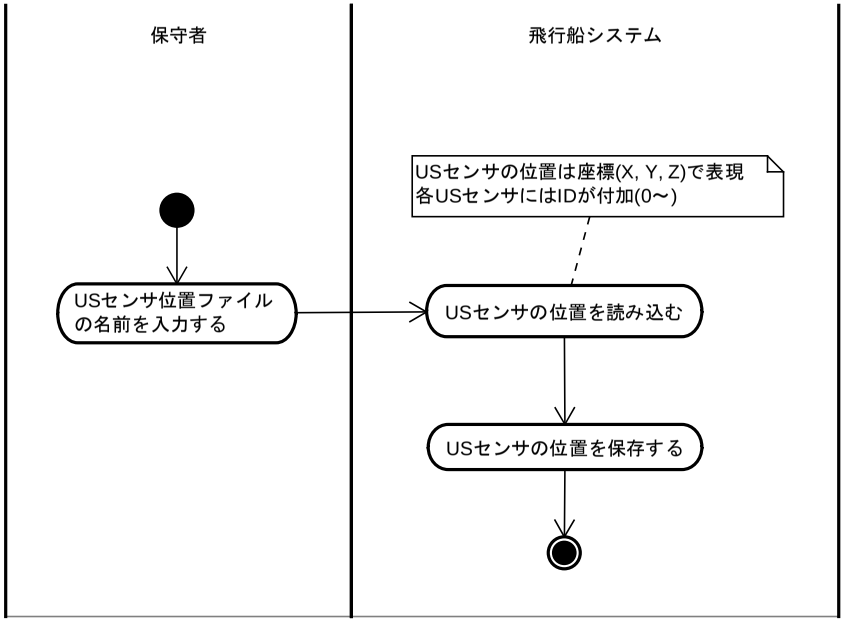

飛行船システム
経路は目的地のリスト
飛行経路を与える
モータの出力値を
決定する
<<include>>
飛行船を自動操縦
で飛ばす
<<include>>
飛行船の状況を表示する
競技者
モータ出力値を送信して
モータを回す
出力値を得る
モータ
高度情報を通知する
高度センサ
高度情報を受信する
飛行特性を与える
USセンサの位置を与える
モータとセンサの試験を行う
保守者
位置情報を通知する
位置センサ
位置情報を受信する
方向情報を通知する
方向センサ
方向情報を受信する
システムを停止させる
目的地を切り替える
タイマ

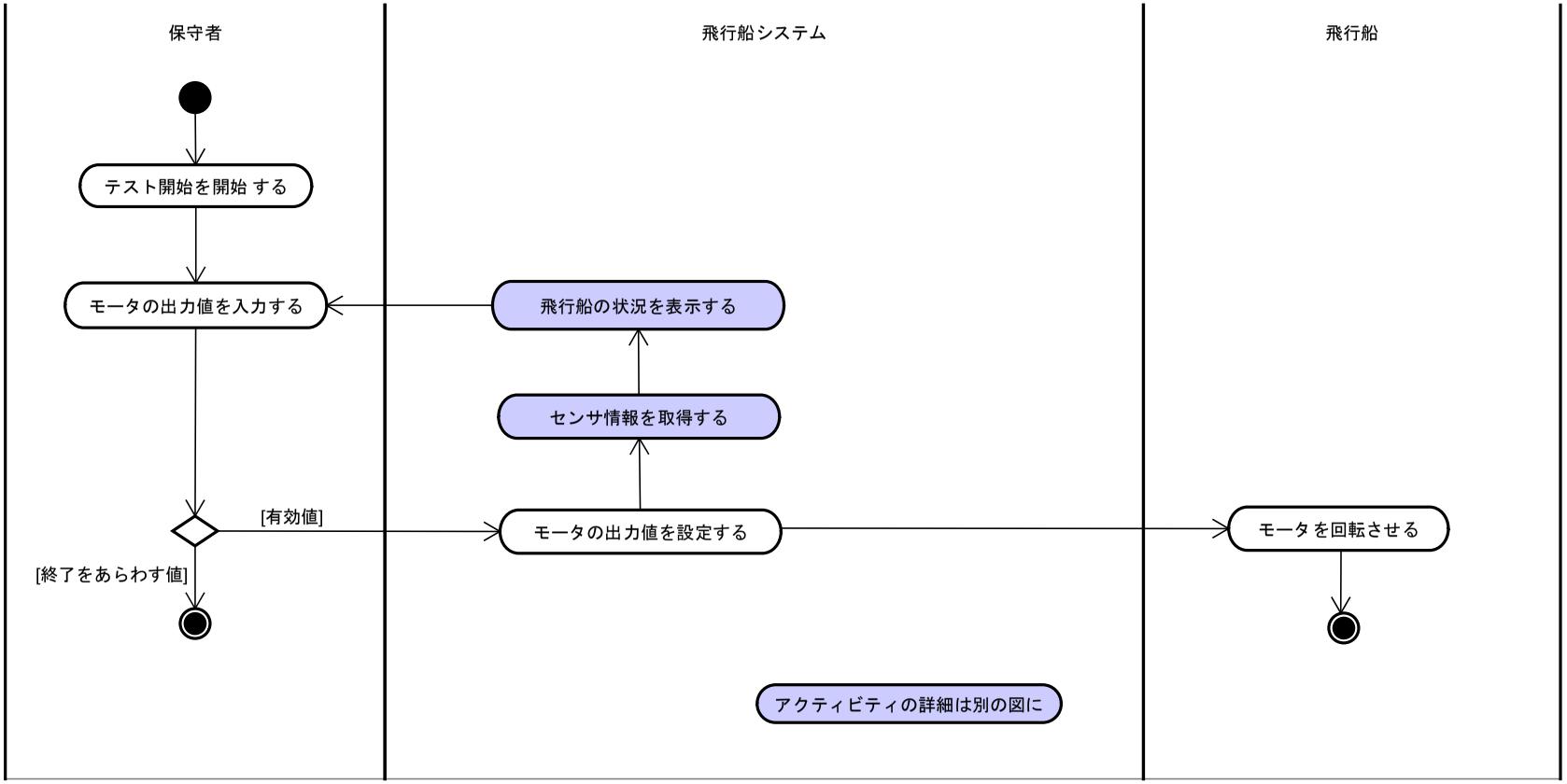

<<entity>>
目的地
*
<<ordered>>
<<entity>>
飛行経路
切り替える
タイマ
1
設定する
設定する
<<device>>
高度センサ
1
飛行船
飛行船システム
<<device>>
方向センサ
1
1
<<device>>
位置センサ
1
* *
* *
飛ばす
1
<<role>>
競技者
1
利用する
<<device>>
モータ
1
1
テストする
利用する
<<entity>>
飛行特性
1
<<role>>
保守者
1
入力する

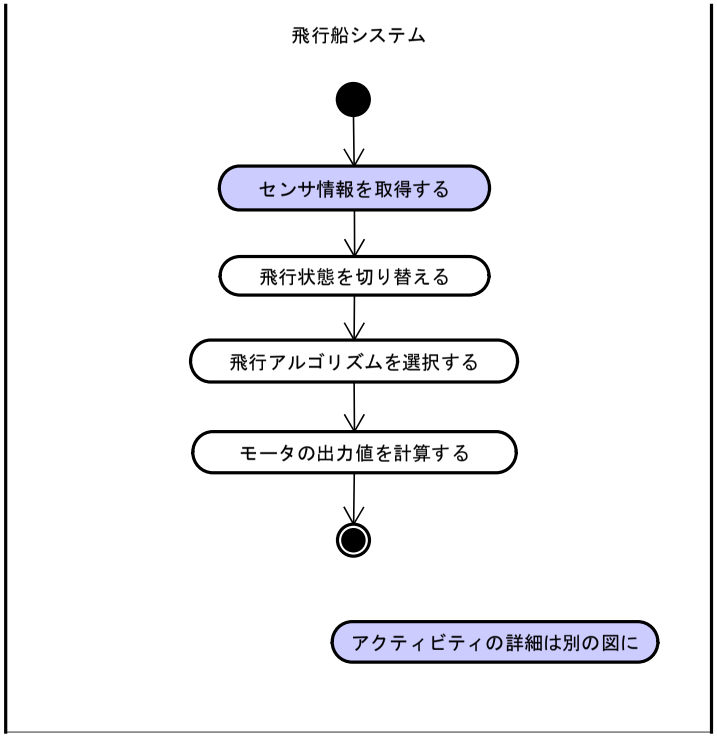

競技者
飛行船システム
目的地は座標(X, Y, Z)で与える
目的地までの予測時間も与える
飛行経路ファイルの名前を入力する
飛行経路を読み込む
飛行経路を保存する

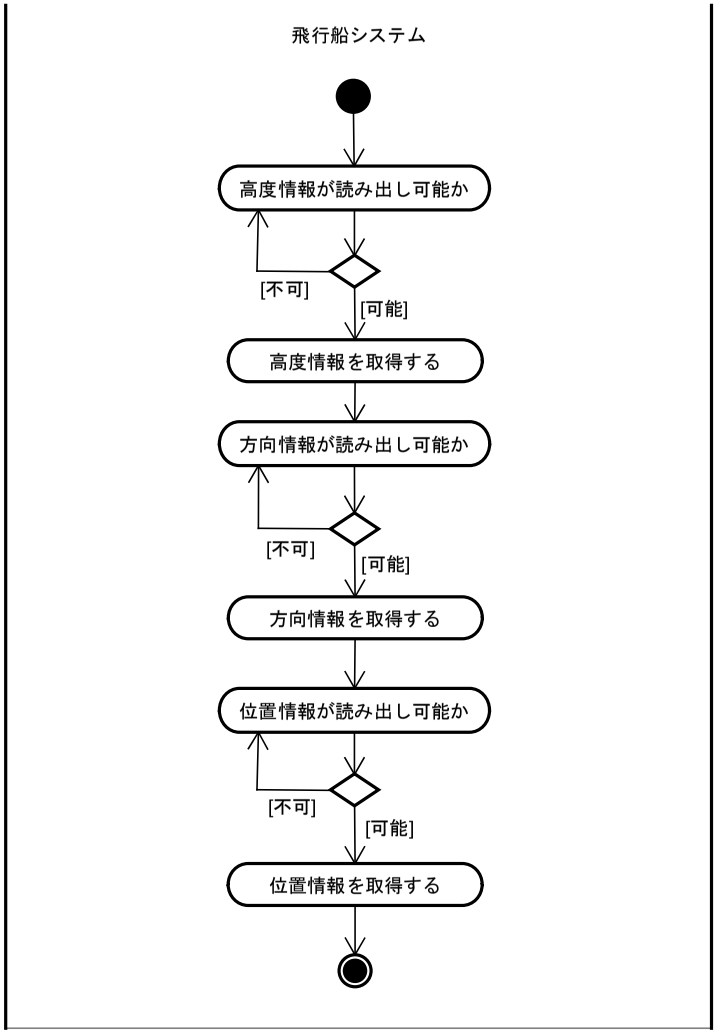

参加者
飛行船システム
飛行船
タイマ
飛行開始を選択する
最初の目的地を設定する
次の目的地を設定する
予測飛行時間を設定する
割り込み時間を設定する
モータの出力値を決定する
タイマを開始する
モータの出力値を設定する
命令を実行する
割り込み時間まで経過したか
飛行船の状況を表示する
[経過していない]
[経過]
[未経過]
目的地に到着したか
予測飛行時間まで経過したか
[未到着]
[到着]
予測飛行時間が経過したことを記録する
予測飛行時間が経過したことを通知する
[経過]
最終目的地か
[最終目的地でない]
[最終目的地]
モータの出力値(停止)を設定する
モータを停止する
着陸メッセージを表示
アクティビティの詳細は別の図に

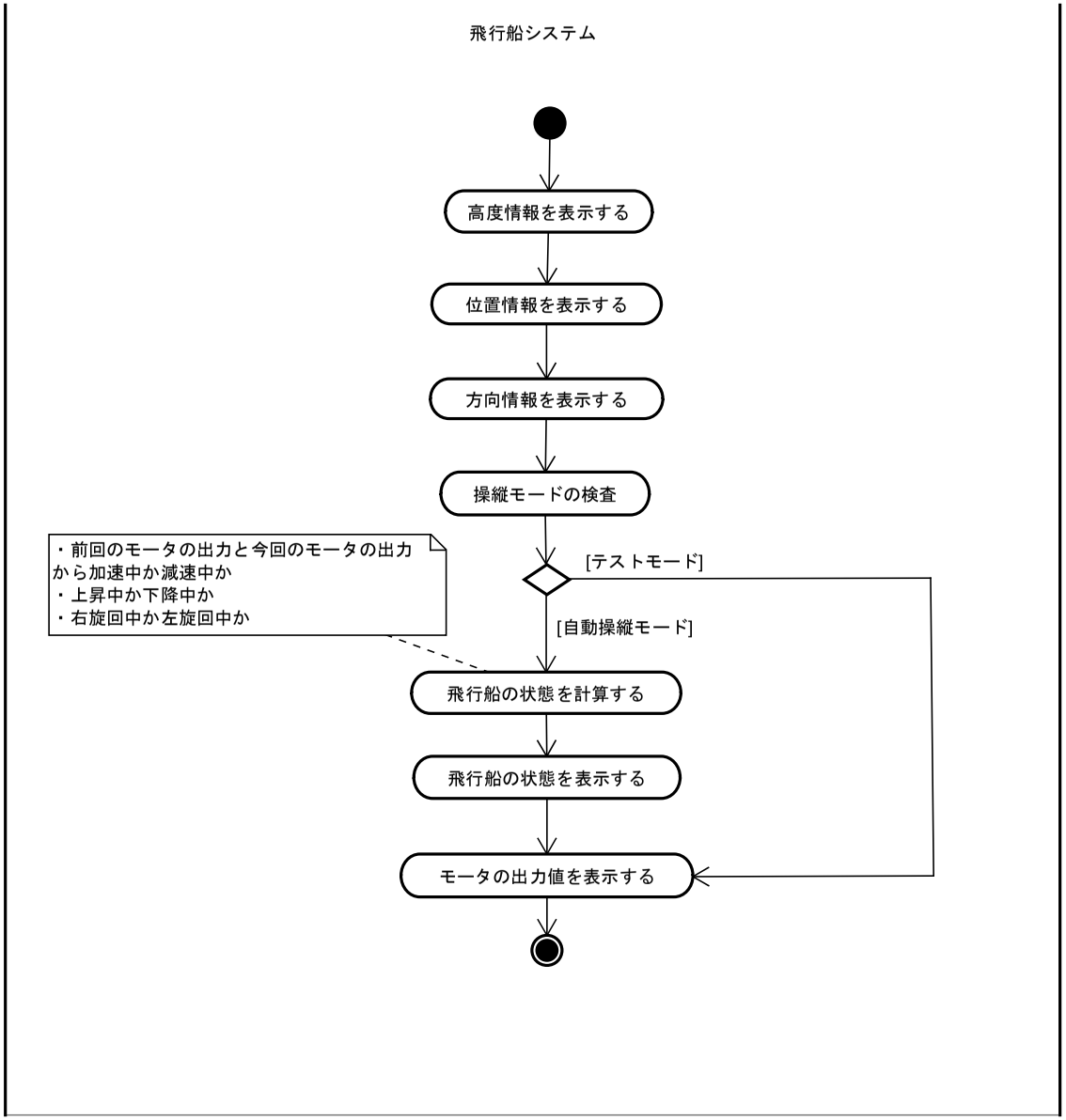

保守者
飛行船システム
飛行特性は，現在地と目的地から算出した距離，角度，高度差に対する表で設定する．表は，飛行モードに対して1つずつ用意する．
飛行特性ファイルの名前を入力する
飛行特性を読み込む
飛行特性を保存する

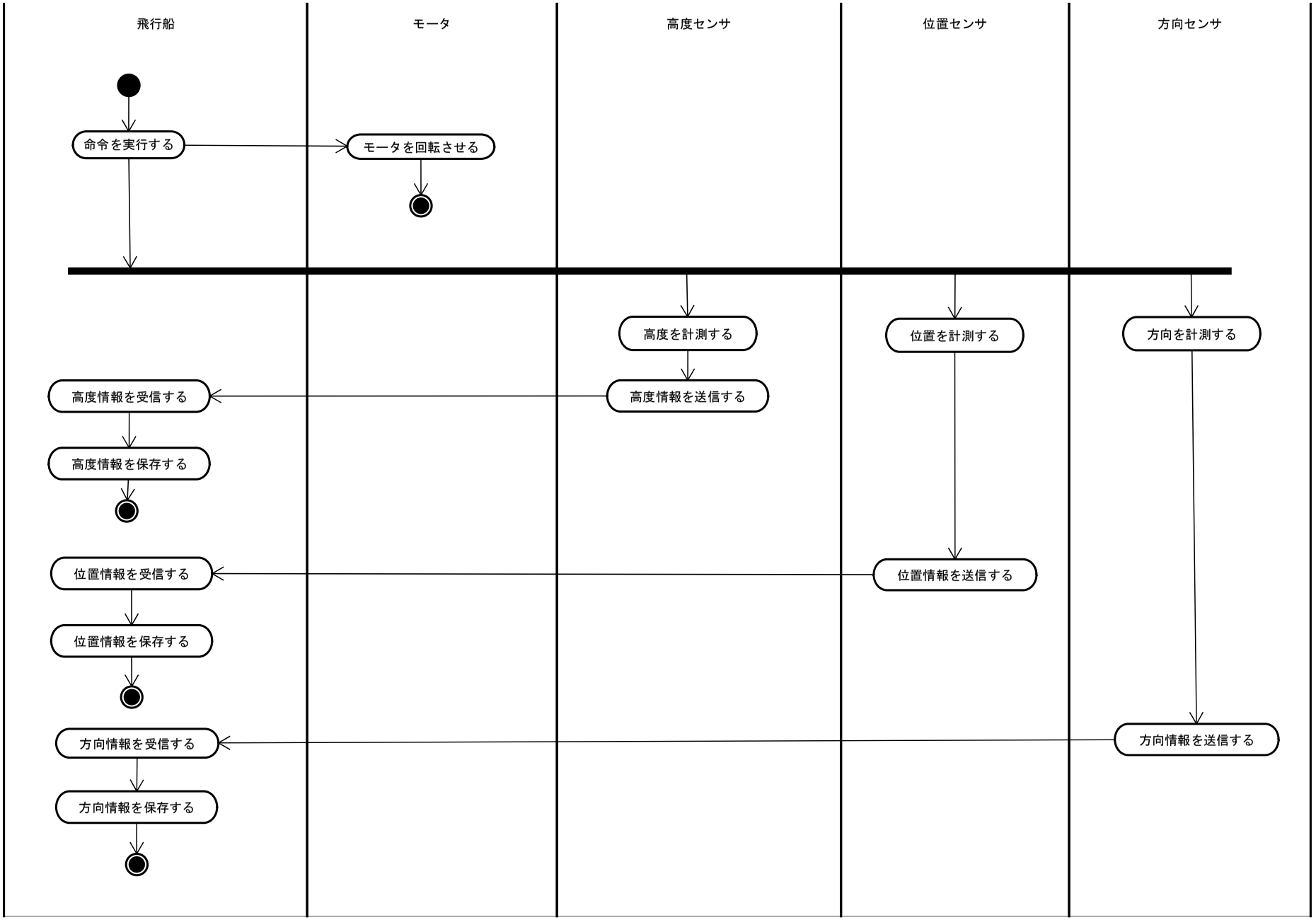

保守者
飛行船システム
USセンサの位置は座標(X, Y, Z)で表現
各USセンサにはIDが付加(0〜)
USセンサ位置ファイル
の名前を入力する
USセンサの位置を読み込む
USセンサの位置を保存する

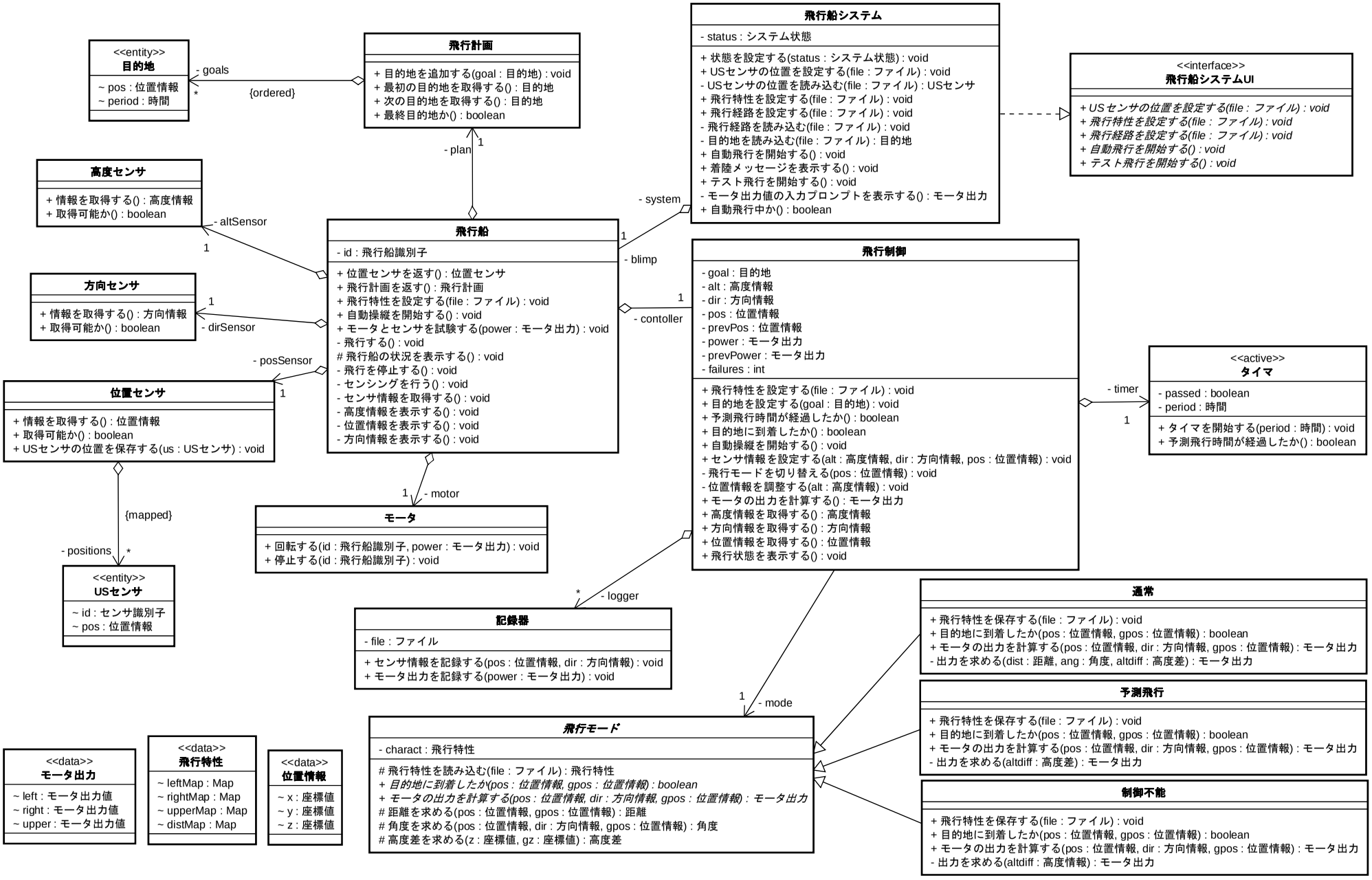

保守者
飛行船システム
飛行船
テスト開始を開始 する
モータの出力値を入力する
飛行船の状況を表示する
センサ情報を取得する
[有効値]
モータの出力値を設定する
モータを回転させる
[終了をあらわす値]
アクティビティの詳細は別の図に

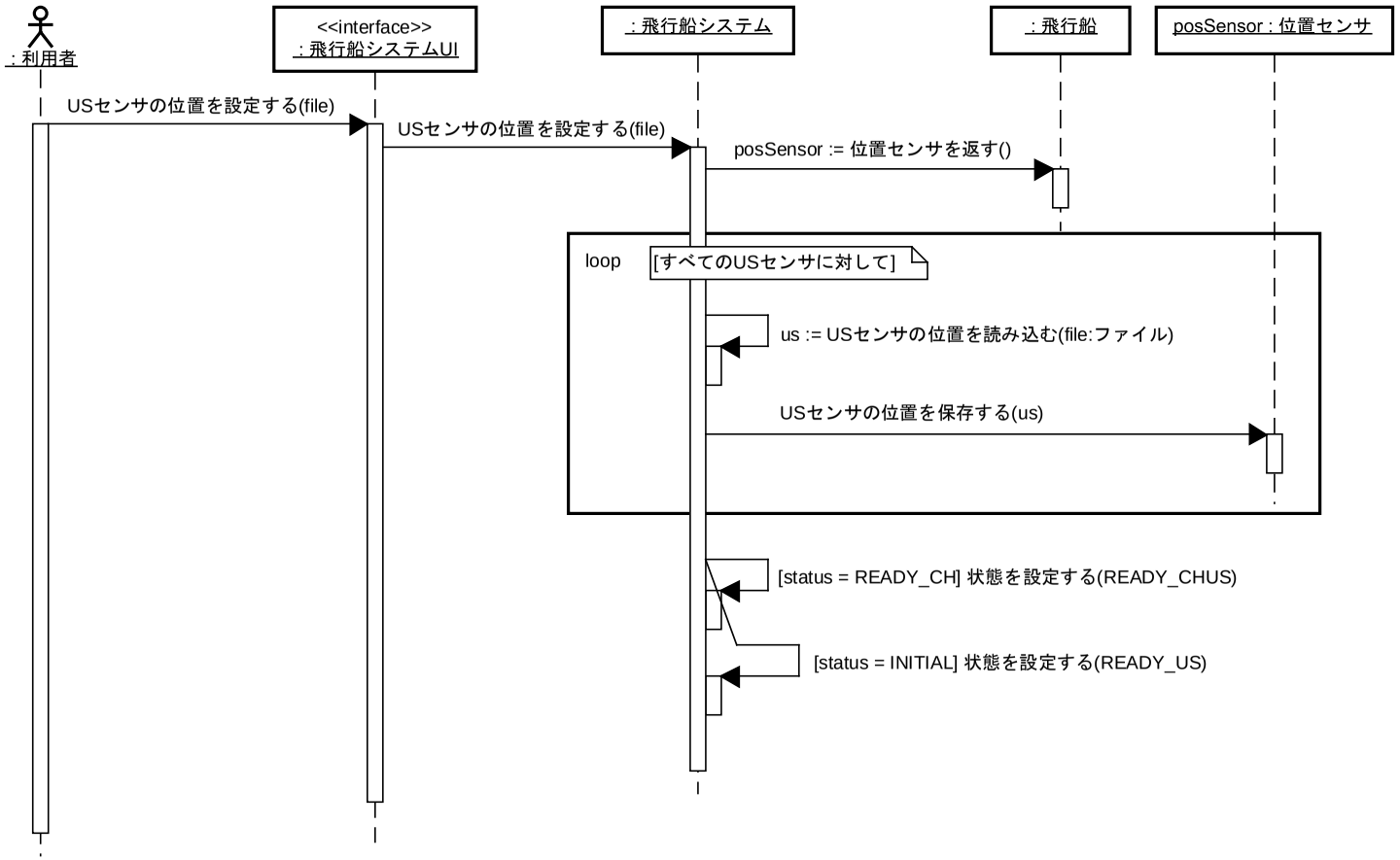

飛行船システム
センサ情報を取得する
飛行状態を切り替える
飛行アルゴリズムを選択する
モータの出力値を計算する
アクティビティの詳細は別の図に

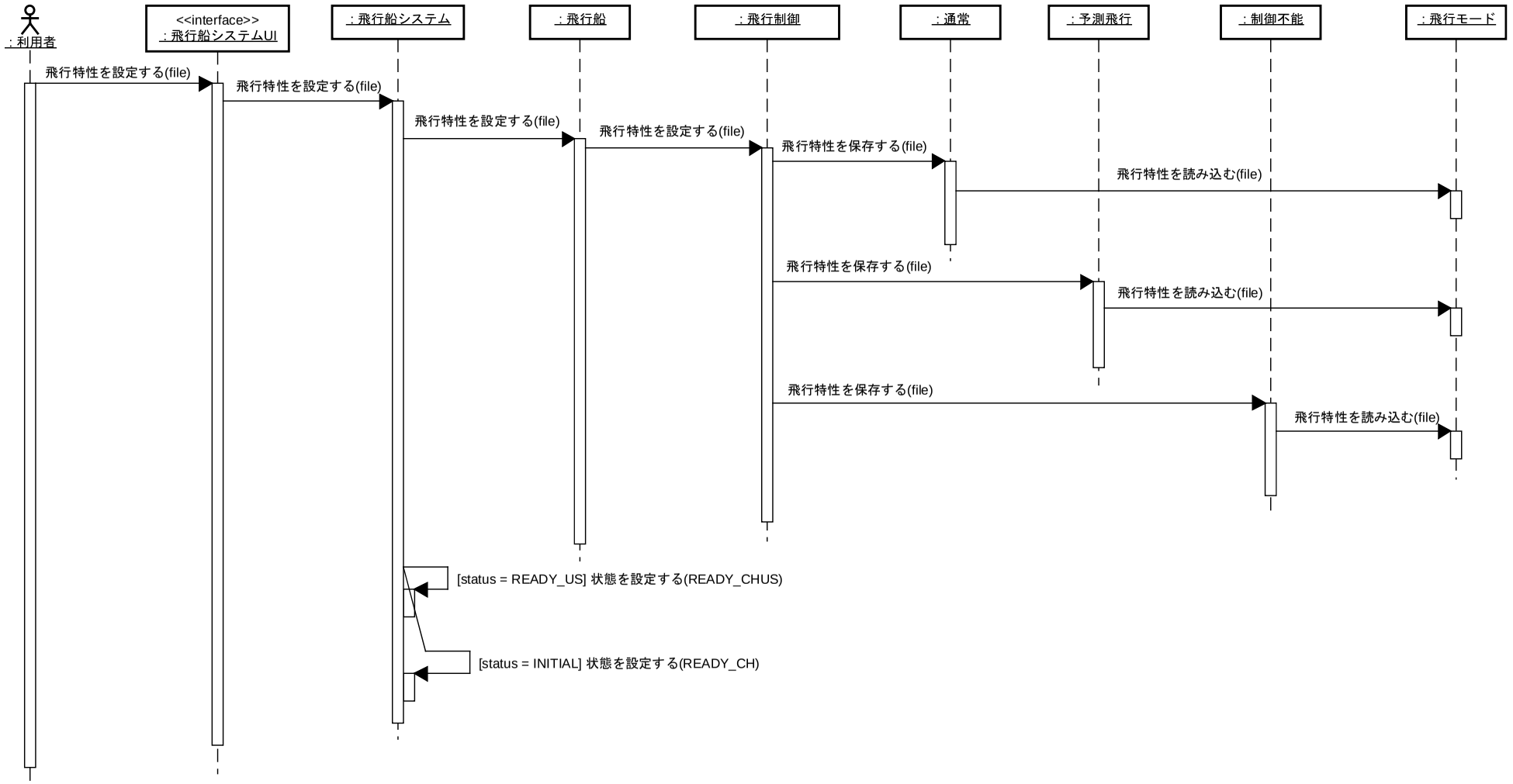

飛行船システム
高度情報が読み出し可能か
[不可]
[可能]
高度情報を取得する
方向情報が読み出し可能か
[不可]
[可能]
方向情報を取得する
位置情報が読み出し可能か
[不可]
[可能]
位置情報を取得する

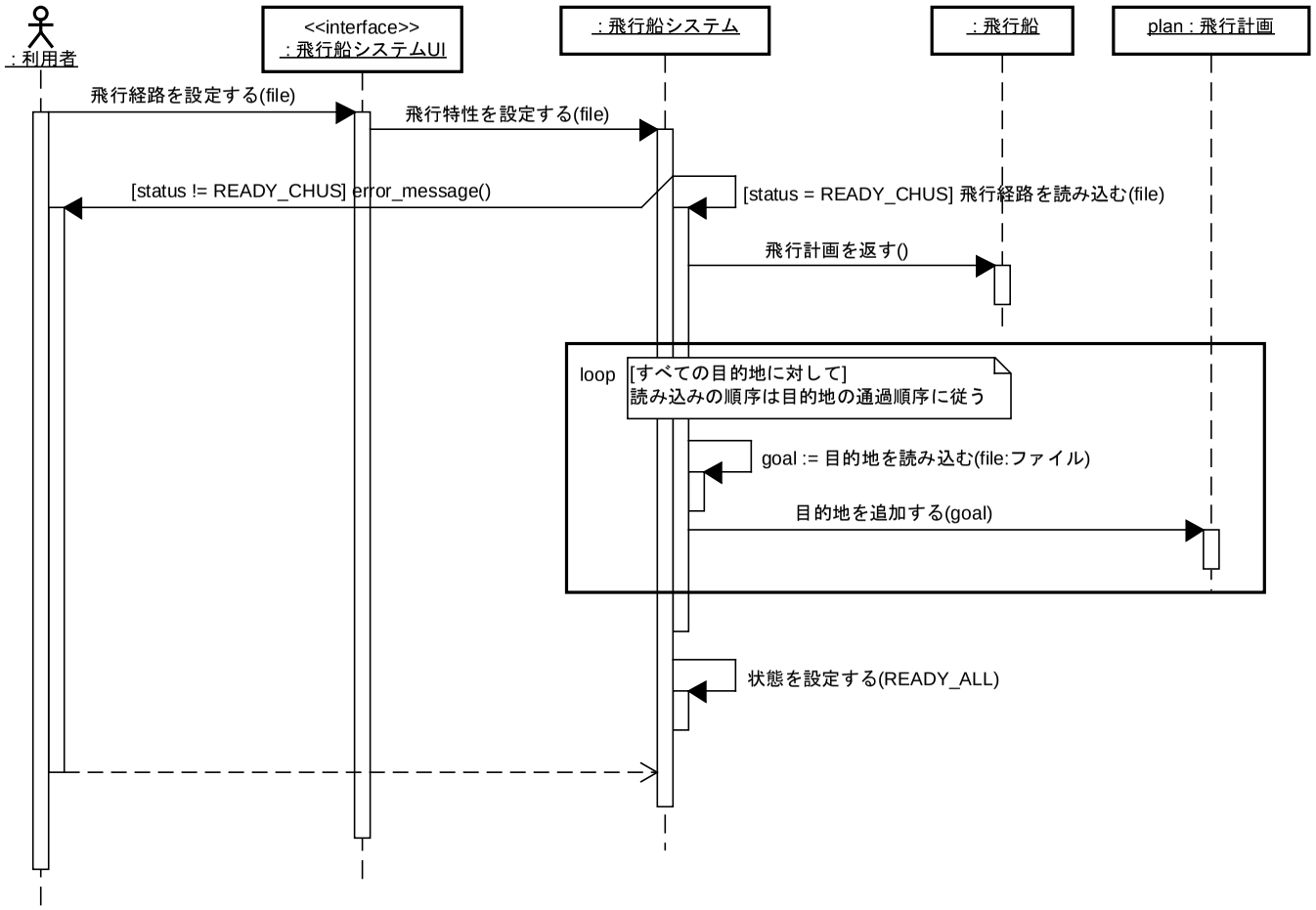

飛行船システム
高度情報を表示する
位置情報を表示する
方向情報を表示する
操縦モードの検査
・前回のモータの出力と今回のモータの出力から加速中か減速中か
・上昇中か下降中か
・右旋回中か左旋回中か
[テストモード]
[自動操縦モード]
飛行船の状態を計算する
飛行船の状態を表示する
モータの出力値を表示する

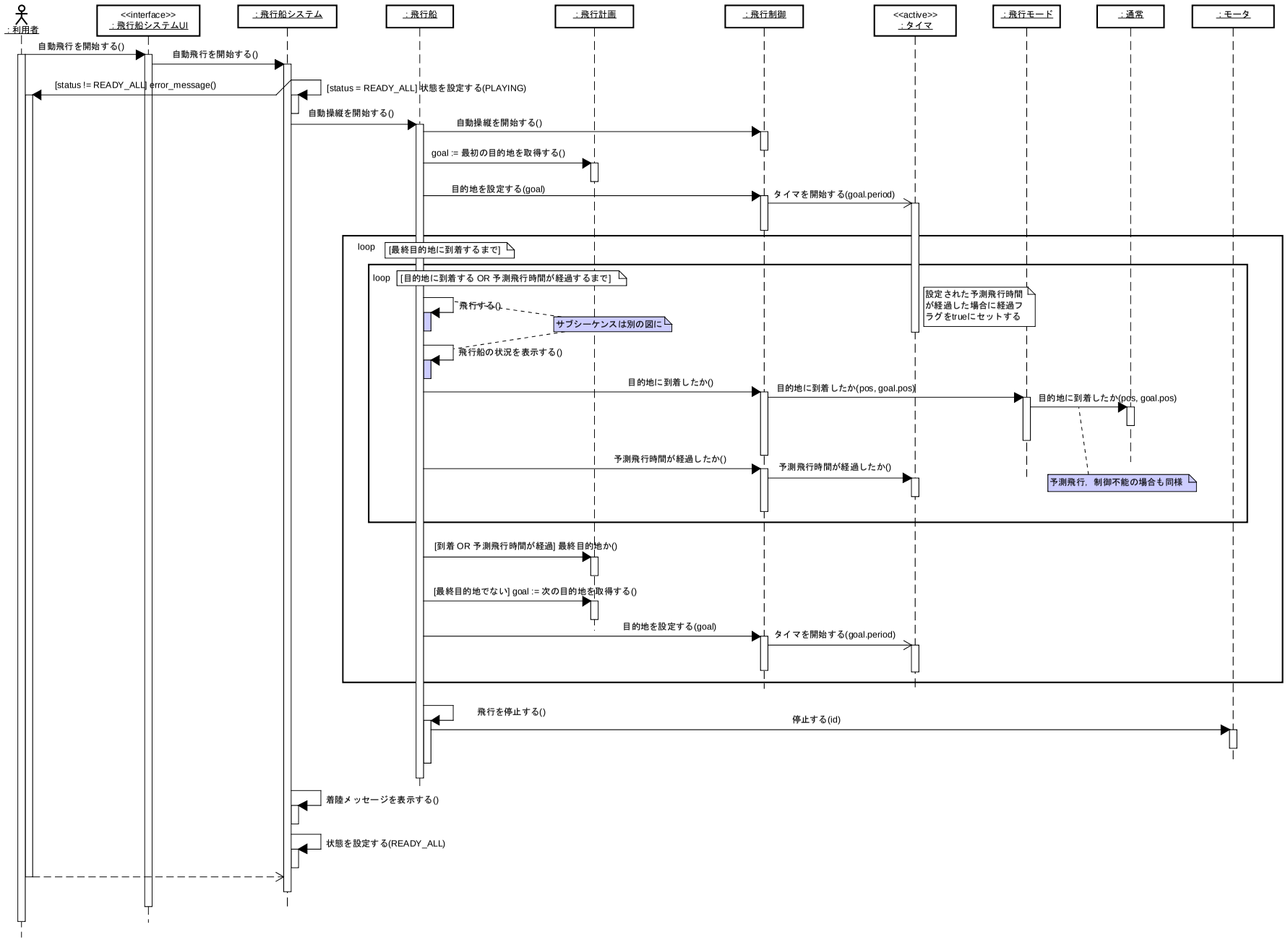

飛行船
モータ
高度センサ
位置センサ
方向センサ
命令を実行する
モータを回転させる
高度を計測する
位置を計測する
方向を計測する
高度情報を受信する
高度情報を送信する
高度情報を保存する
位置情報を受信する
位置情報を送信する
位置情報を保存する
方向情報を受信する
方向情報を送信する
方向情報を保存する


<<entity>>
目的地
~ pos : 位置情報
~ period : 時間
- goals
*
{ordered}
飛行計画
+ 目的地を追加する(goal : 目的地) : void
+ 最初の目的地を取得する() : 目的地
+ 次の目的地を取得する() : 目的地
+ 最終目的地か() : boolean
- plan
1
飛行船システム
- status : システム状態
+ 状態を設定する(status : システム状態) : void
+ USセンサの位置を設定する(file : ファイル) : void
- USセンサの位置を読み込む(file : ファイル) : USセンサ
+ 飛行特性を設定する(file : ファイル) : void
+ 飛行経路を設定する(file : ファイル) : void
- 飛行経路を読み込む(file : ファイル) : void
- 目的地を読み込む(file : ファイル) : 目的地
+ 自動飛行を開始する() : void
+ 着陸メッセージを表示する() : void
+ テスト飛行を開始する() : void
- モータ出力値の入力プロンプトを表示する() : モータ出力
+ 自動飛行中か() : boolean
<<interface>>
飛行船システムUI
+ USセンサの位置を設定する(file : ファイル) : void
+ 飛行特性を設定する(file : ファイル) : void
+ 飛行経路を設定する(file : ファイル) : void
+ 自動飛行を開始する() : void
+ テスト飛行を開始する() : void
- system
高度センサ
+ 情報を取得する() : 高度情報
+ 取得可能か() : boolean
- altSensor
1
飛行船
- id : 飛行船識別子
+ 位置センサを返す() : 位置センサ
+ 飛行計画を返す() : 飛行計画
+ 飛行特性を設定する(file : ファイル) : void
+ 自動操縦を開始する() : void
+ モータとセンサを試験する(power : モータ出力) : void
- 飛行する() : void
飛行船の状況を表示する() : void
- 飛行を停止する() : void
- センシングを行う() : void
- センサ情報を取得する() : void
- 高度情報を表示する() : void
- 位置情報を表示する() : void
- 方向情報を表示する() : void
1
- blimp
飛行制御
- goal : 目的地
- alt : 高度情報
- dir : 方向情報
- pos : 位置情報
- prevPos : 位置情報
- power : モータ出力
- prevPower : モータ出力
- failures : int
+ 飛行特性を設定する(file : ファイル) : void
+ 目的地を設定する(goal : 目的地) : void
+ 予測飛行時間が経過したか() : boolean
+ 目的地に到着したか() : boolean
+ 自動操縦を開始する() : void
+ センサ情報を設定する(alt : 高度情報, dir : 方向情報, pos : 位置情報) : void
- 飛行モードを切り替える(pos : 位置情報) : void
- 位置情報を調整する(alt : 高度情報) : void
+ モータの出力を計算する() : モータ出力
+ 高度情報を取得する() : 高度情報
+ 方向情報を取得する() : 方向情報
+ 位置情報を取得する() : 位置情報
+ 飛行状態を表示する() : void
1
- contoller
方向センサ
+ 情報を取得する() : 方向情報
+ 取得可能か() : boolean
1
- dirSensor
- posSensor
位置センサ
+ 情報を取得する() : 位置情報
+ 取得可能か() : boolean
+ USセンサの位置を保存する(us : USセンサ) : void
1
- timer
1
<<active>>
タイマ
- passed : boolean
- period : 時間
+ タイマを開始する(period : 時間) : void
+ 予測飛行時間が経過したか() : boolean
{mapped}
- positions
*
<<entity>>
USセンサ
~ id : センサ識別子
~ pos : 位置情報
1
- motor
モータ
+ 回転する(id : 飛行船識別子, power : モータ出力) : void
+ 停止する(id : 飛行船識別子) : void
*
- logger
記録器
- file : ファイル
+ センサ情報を記録する(pos : 位置情報, dir : 方向情報) : void
+ モータ出力を記録する(power : モータ出力) : void
1
- mode
飛行モード
- charact : 飛行特性
飛行特性を読み込む(file : ファイル) : 飛行特性
+ 目的地に到着したか(pos : 位置情報, gpos : 位置情報) : boolean
+ モータの出力を計算する(pos : 位置情報, dir : 方向情報, gpos : 位置情報) : モータ出力
# 距離を求める(pos : 位置情報, gpos : 位置情報) : 距離
# 角度を求める(pos : 位置情報, dir : 方向情報, gpos : 位置情報) : 角度
# 高度差を求める(z : 座標値, gz : 座標値) : 高度差
通常
+ 飛行特性を保存する(file : ファイル) : void
+ 目的地に到着したか(pos : 位置情報, gpos : 位置情報) : boolean
+ モータの出力を計算する(pos : 位置情報, dir : 方向情報, gpos : 位置情報) : モータ出力
- 出力を求める(dist : 距離, ang : 角度, altdiff : 高度差) : モータ出力
予測飛行
+ 飛行特性を保存する(file : ファイル) : void
+ 目的地に到着したか(pos : 位置情報, gpos : 位置情報) : boolean
+ モータの出力を計算する(pos : 位置情報, dir : 方向情報, gpos : 位置情報) : モータ出力
- 出力を求める(altdiff : 高度差) : モータ出力
制御不能
+ 飛行特性を保存する(file : ファイル) : void
+ 目的地に到着したか(pos : 位置情報, gpos : 位置情報) : boolean
+ モータの出力を計算する(pos : 位置情報, dir : 方向情報, gpos : 位置情報) : モータ出力
- 出力を求める(altdiff : 高度情報) : モータ出力
<<data>>
モータ出力
~ left : モータ出力値
~ right : モータ出力値
~ upper : モータ出力値
<<data>>
飛行特性
~ leftMap : Map
~ rightMap : Map
~ upperMap : Map
~ distMap : Map
<<data>>
位置情報
~ x : 座標値
~ y : 座標値
~ z : 座標値

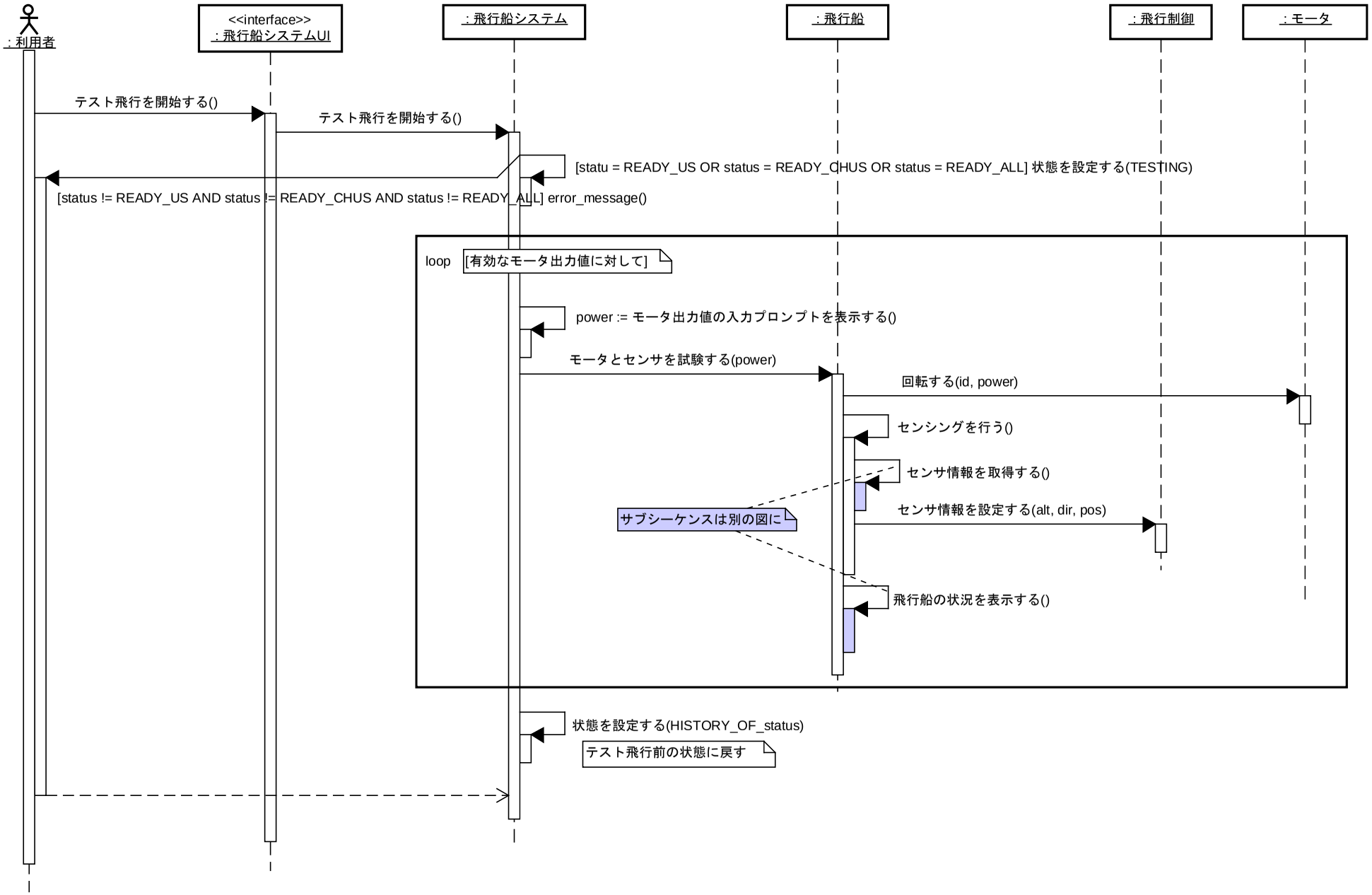

：利用者
<<interface>>
：飛行船システムUI
：飛行船システム
：飛行船
posSensor : 位置センサ
USセンサの位置を設定する(file)
USセンサの位置を設定する(file)
posSensor := 位置センサを返す()
loop
[すべてのUSセンサに対して]
us := USセンサの位置を読み込む(file:ファイル)
USセンサの位置を保存する(us)
[status = READY_CH] 状態を設定する(READY_CHUS)
[status = INITIAL] 状態を設定する(READY_US)

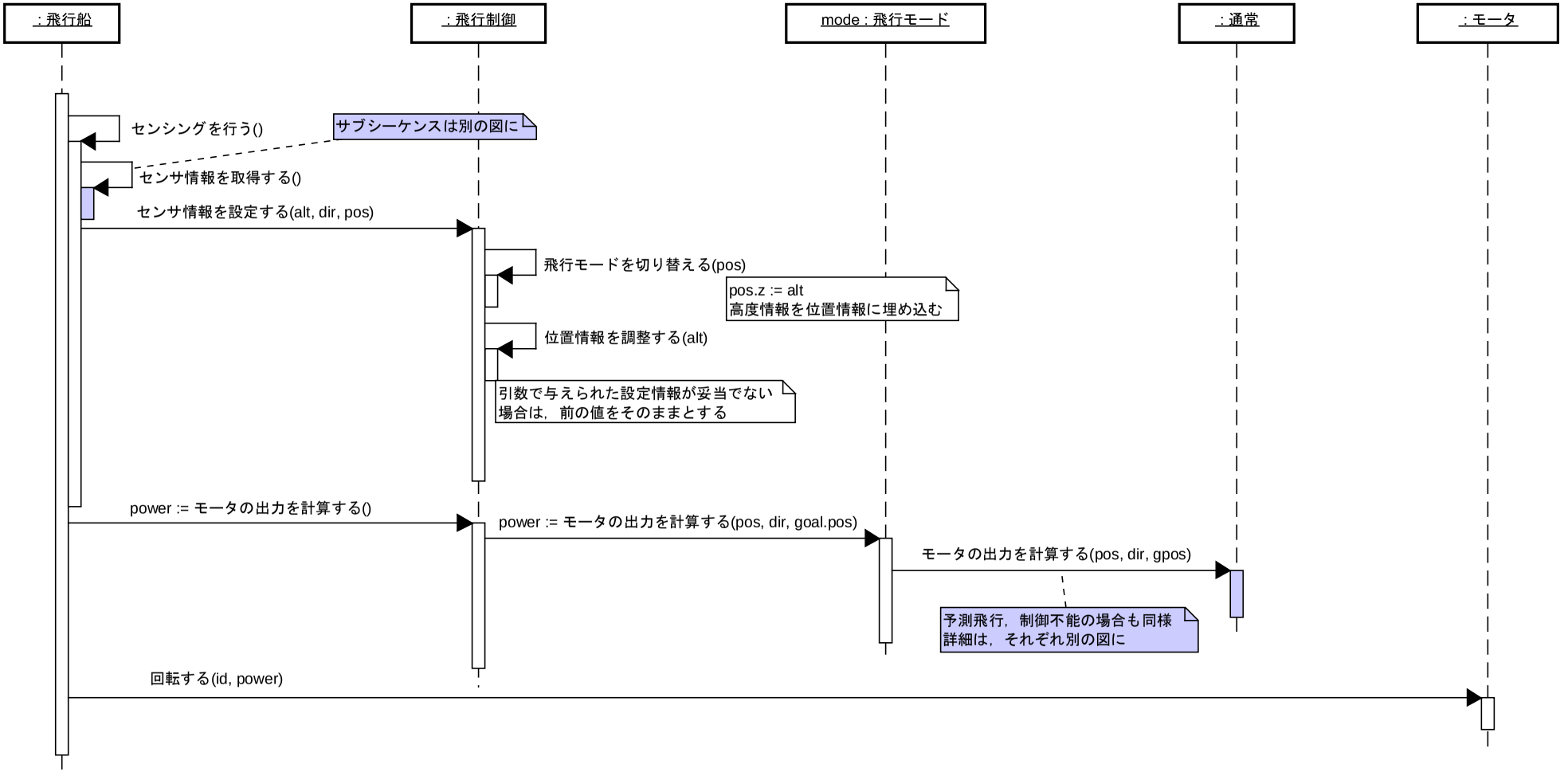

：利用者
<<interface>>
：飛行船システムUI
：飛行船システム
：飛行船
：飛行制御
：通常
：予測飛行
：制御不能
：飛行モード
飛行特性を設定する(file)
飛行特性を設定する(file)
飛行特性を設定する(file)
飛行特性を設定する(file)
飛行特性を保存する(file)
飛行特性を読み込む(file)
飛行特性を保存する(file)
飛行特性を読み込む(file)
飛行特性を保存する(file)
飛行特性を読み込む(file)
[status = READY_US] 状態を設定する(READY_CHUS)
[status = INITIAL] 状態を設定する(READY_CH)

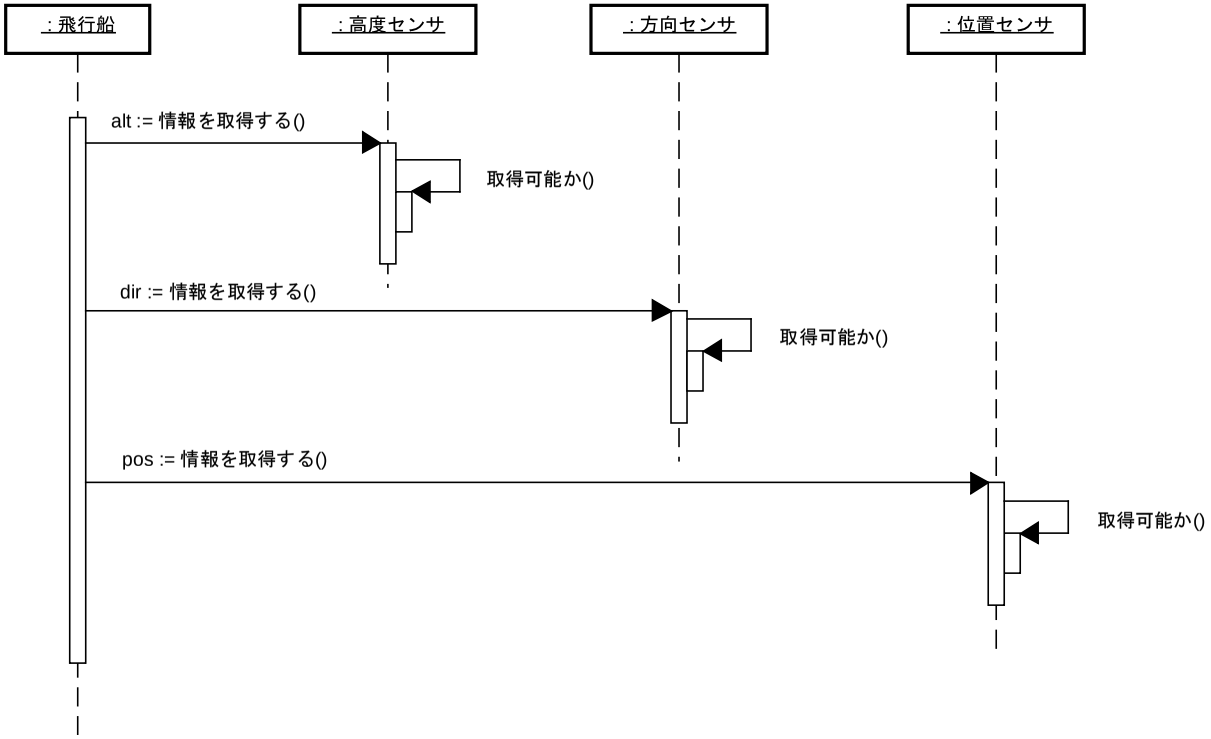

：利用者
<<interface>>
：飛行船システムUI
：飛行船システム
：飛行船
plan：飛行計画
飛行経路を設定する(file)
飛行特性を設定する(file)
[status != READY_CHUS] error_message()
[status = READY_CHUS] 飛行経路を読み込む(file)
飛行計画を返す()
loop
[すべての目的地に対して]
読み込みの順序は目的地の通過順序に従う
goal := 目的地を読み込む(file:ファイル)
目的地を追加する(goal)
状態を設定する(READY_ALL)

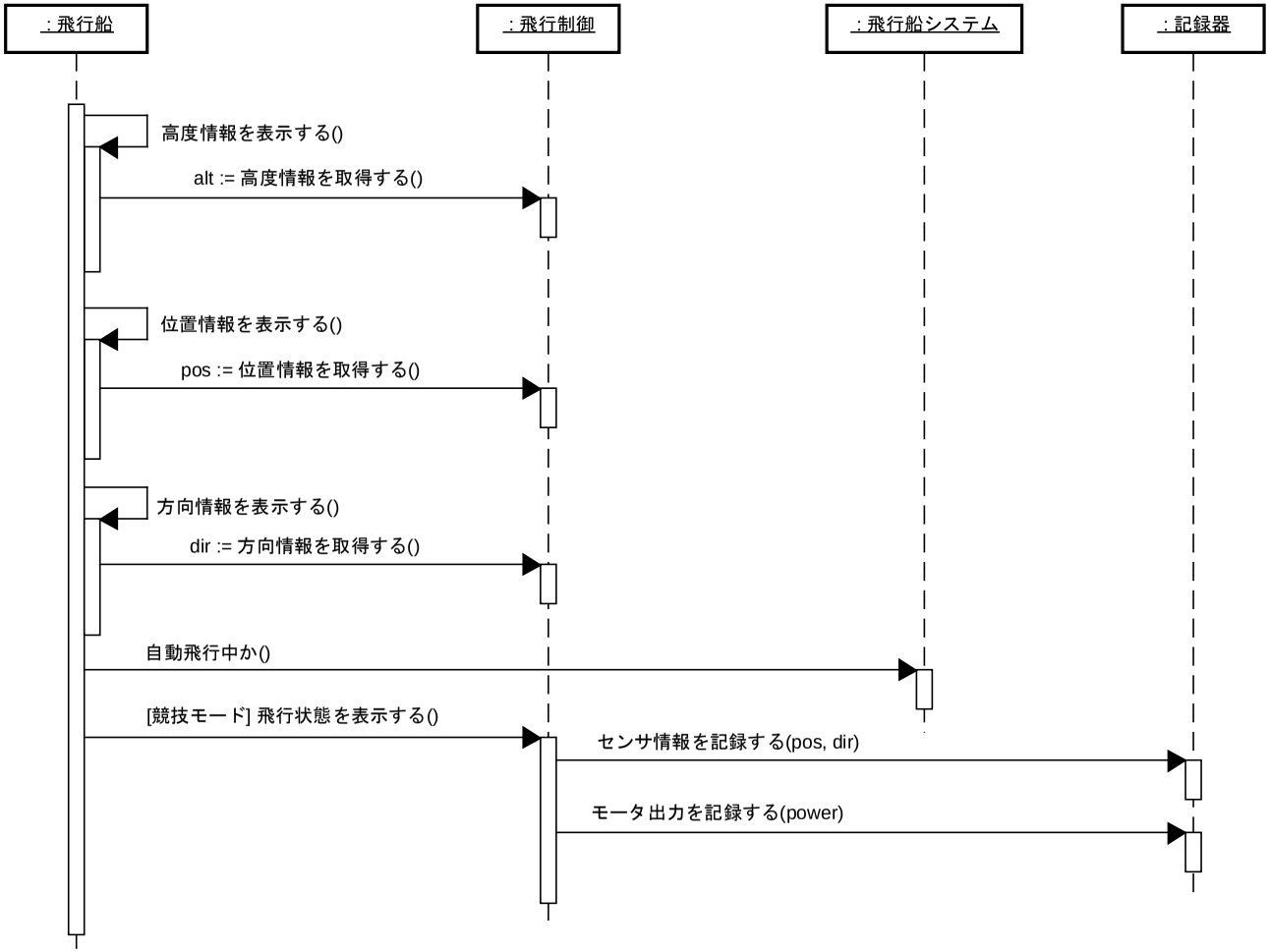

：利用者
<<interface>>
：飛行船システムUI
：飛行船システム
：飛行船
：飛行計画
：飛行制御
<<active>>
：タイマ
：飛行モード
：通常
：モータ
自動飛行を開始する()
自動飛行を開始する()
[status != READY_ALL] error_message()
[status = READY_ALL] 状態を設定する(PLAYING)
自動操縦を開始する()
自動操縦を開始する()
goal := 最初の目的地を取得する()
目的地を設定する(goal)
タイマを開始する(goal.period)
loop
[最終目的地に到着するまで]
loop
[目的地に到着する OR 予測飛行時間が経過するまで]
設定された予測飛行時間が経過した場合に経過フラグをtrueにセットする
飛行する()
サブシーケンスは別の図に
飛行船の状況を表示する()
目的地に到着したか()
目的地に到着したか(pos, goal.pos)
目的地に到着したか(pos, goal.pos)
予測飛行時間が経過したか()
予測飛行時間が経過したか()
予測飛行，制御不能の場合も同様
[到着 OR 予測飛行時間が経過] 最終目的地か()
[最終目的地でない] goal := 次の目的地を取得する()
目的地を設定する(goal)
タイマを開始する(goal.period)
飛行を停止する()
停止する(id)
着陸メッセージを表示する()
状態を設定する(READY_ALL)

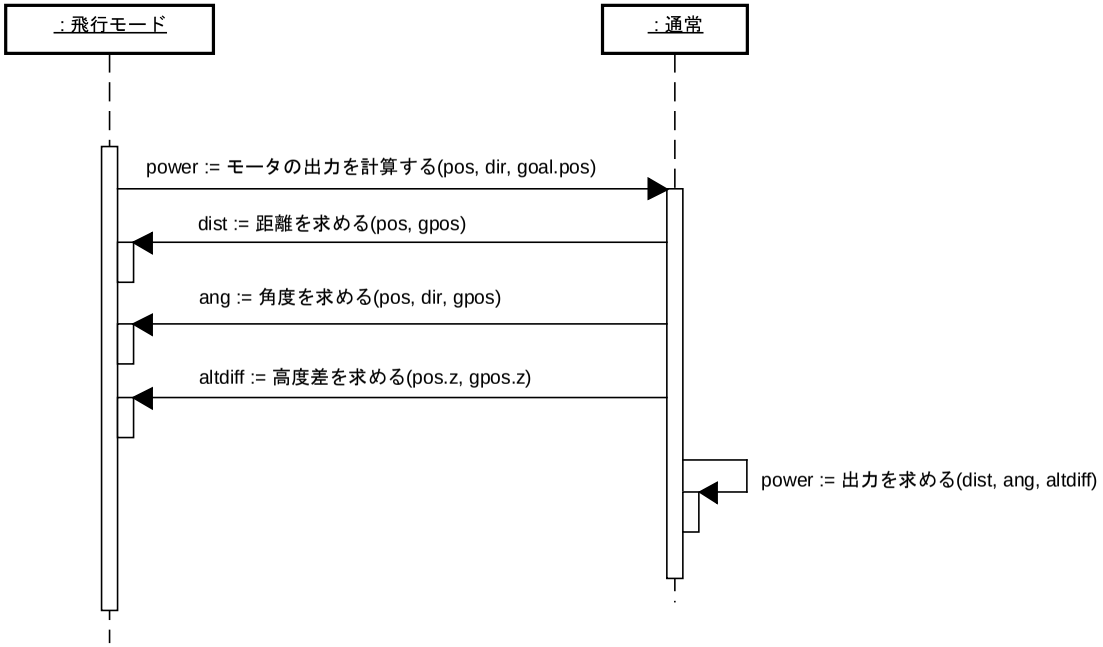
設計モデル(5/15)

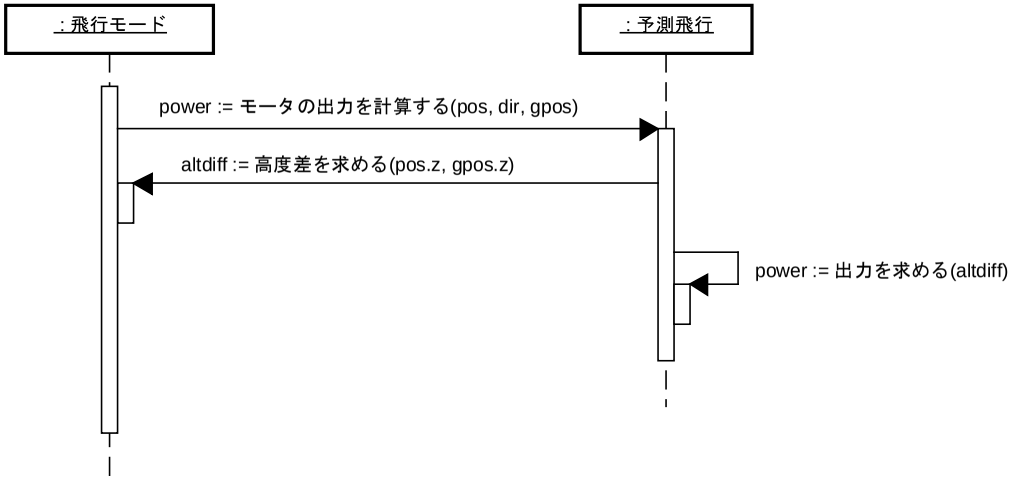

：利用者
<<interface>>
：飛行船システムUI
：飛行船システム
：飛行船
：飛行制御
：モータ
テスト飛行を開始する()
テスト飛行を開始する()
[statu = READY_US OR status = READY_CHUS OR status = READY_ALL] 状態を設定する(TESTING)
[status != READY_US AND status != READY_CHUS AND status != READY_ALL] error_message()
loop
[有効なモータ出力値に対して]
power := モータ出力値の入力プロンプトを表示する()
モータとセンサを試験する(power)
回転する(id, power)
センシングを行う()
センサ情報を取得する()
サブシーケンスは別の図に
センサ情報を設定する(alt, dir, pos)
飛行船の状況を表示する()
状態を設定する(HISTORY_OF_status)
テスト飛行前の状態に戻す

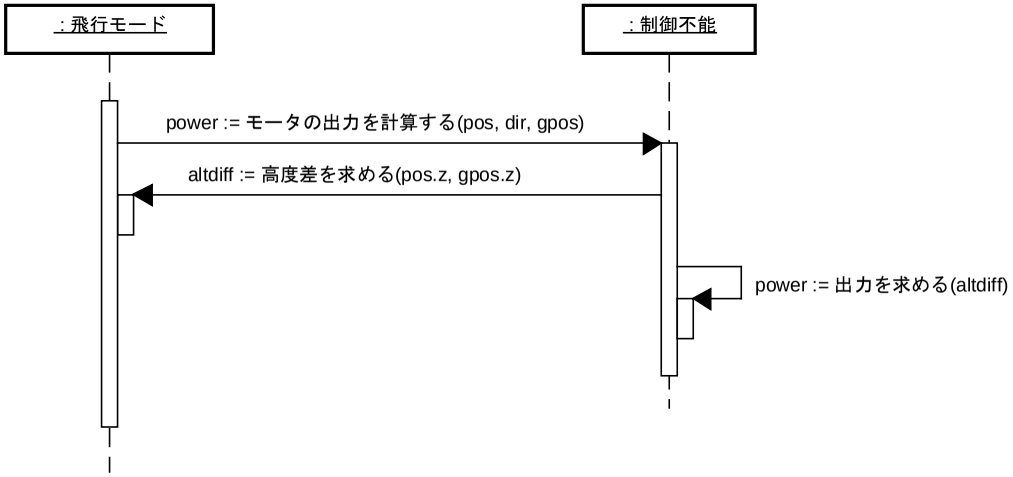

：飛行船
：飛行制御
mode : 飛行モード
：通常
：モータ
センシングを行う()
サブシーケンスは別の図に
センサ情報を取得する()
センサ情報を設定する(alt, dir, pos)
飛行モードを切り替える(pos)
pos.z := alt
高度情報を位置情報に埋め込む
位置情報を調整する(alt)
引数で与えられた設定情報が妥当でない
場合は，前の値をそのままとする
power := モータの出力を計算する()
power := モータの出力を計算する(pos, dir, goal.pos)
モータの出力を計算する(pos, dir, gpos)
予測飛行，制御不能の場合も同様
詳細は，それぞれ別の図に
回転する(id, power)

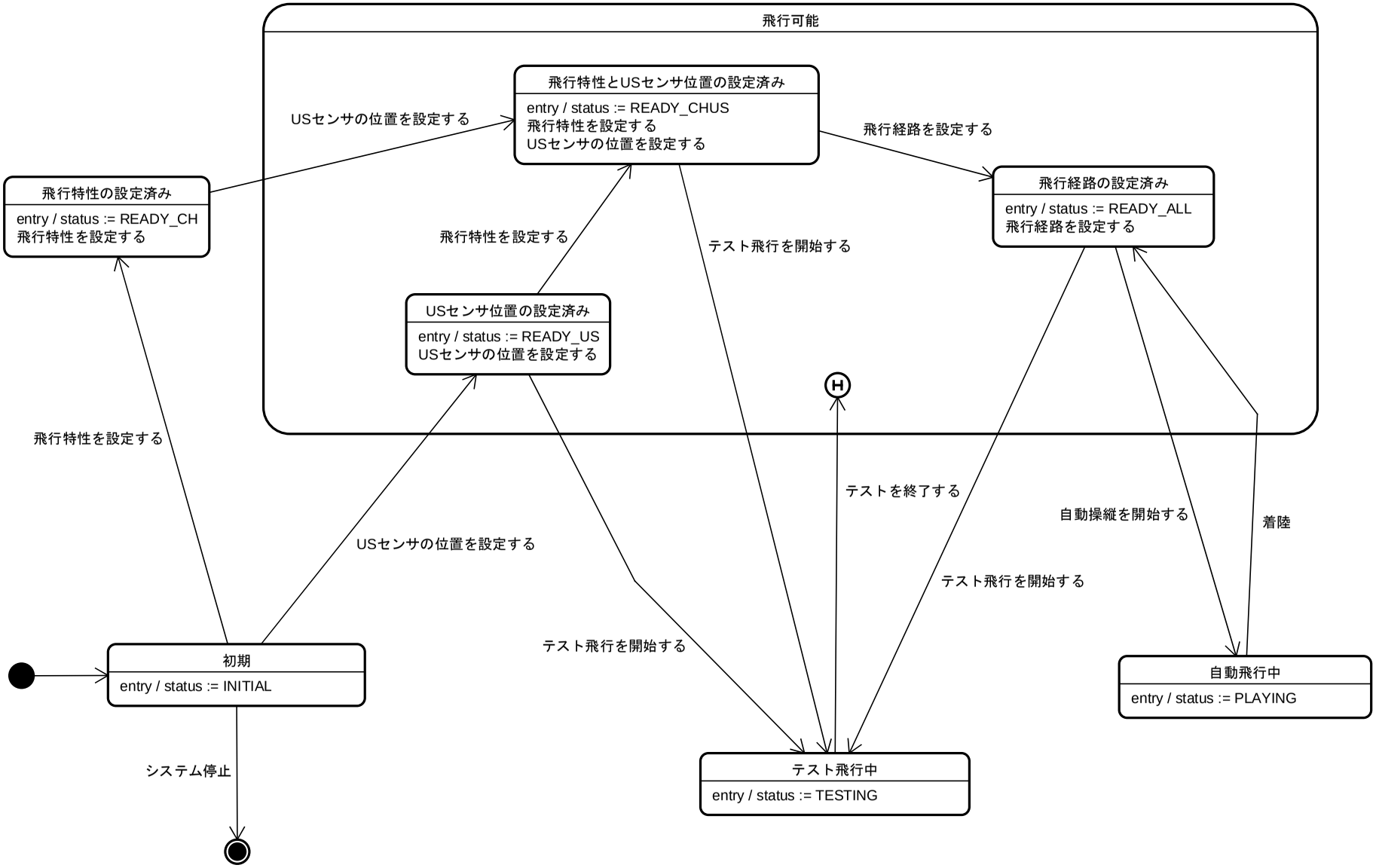

：飛行船
：高度センサ
：方向センサ
：位置センサ
alt := 情報を取得する()
取得可能か()
dir := 情報を取得する()
取得可能か()
pos := 情報を取得する()
取得可能か()

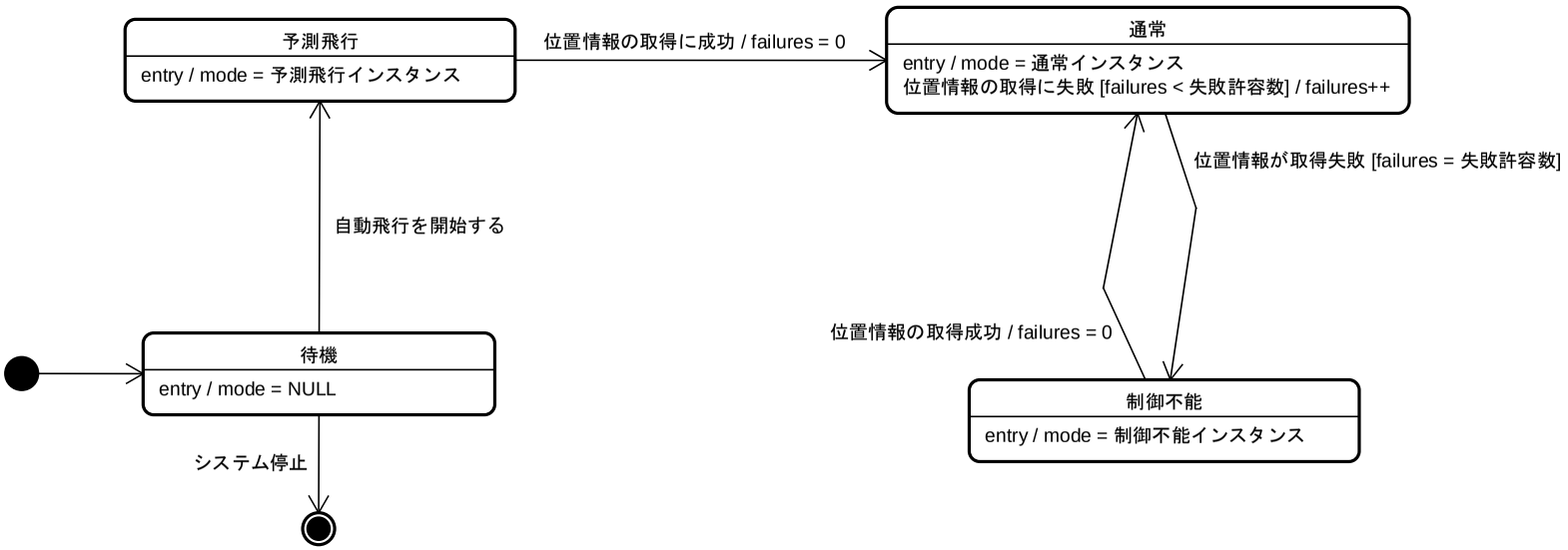

：飛行船
：飛行制御
：飛行船システム
：記録器
高度情報を表示する()
alt := 高度情報を取得する()
位置情報を表示する()
pos := 位置情報を取得する()
方向情報を表示する()
dir := 方向情報を取得する()
自動飛行中か()
[競技モード] 飛行状態を表示する()
センサ情報を記録する(pos, dir)
モータ出力を記録する(power)

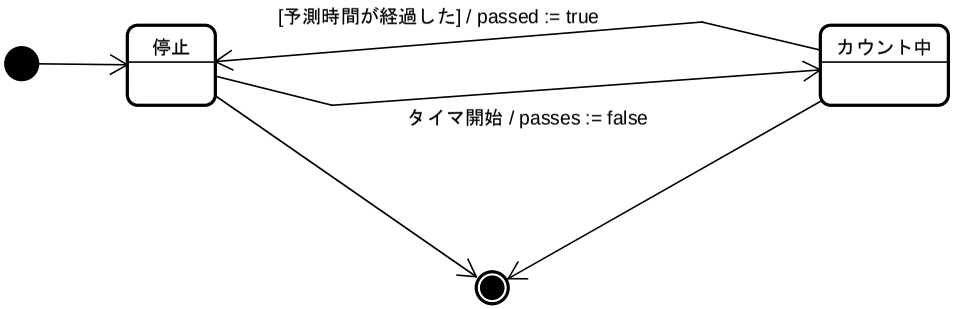

：飛行モード
：通常
power := モータの出力を計算する(pos, dir, goal.pos)
dist := 距離を求める(pos, gpos)
ang := 角度を求める(pos, dir, gpos)
altdiff := 高度差を求める(pos.z, gpos.z)
power := 出力を求める(dist, ang, altdiff)

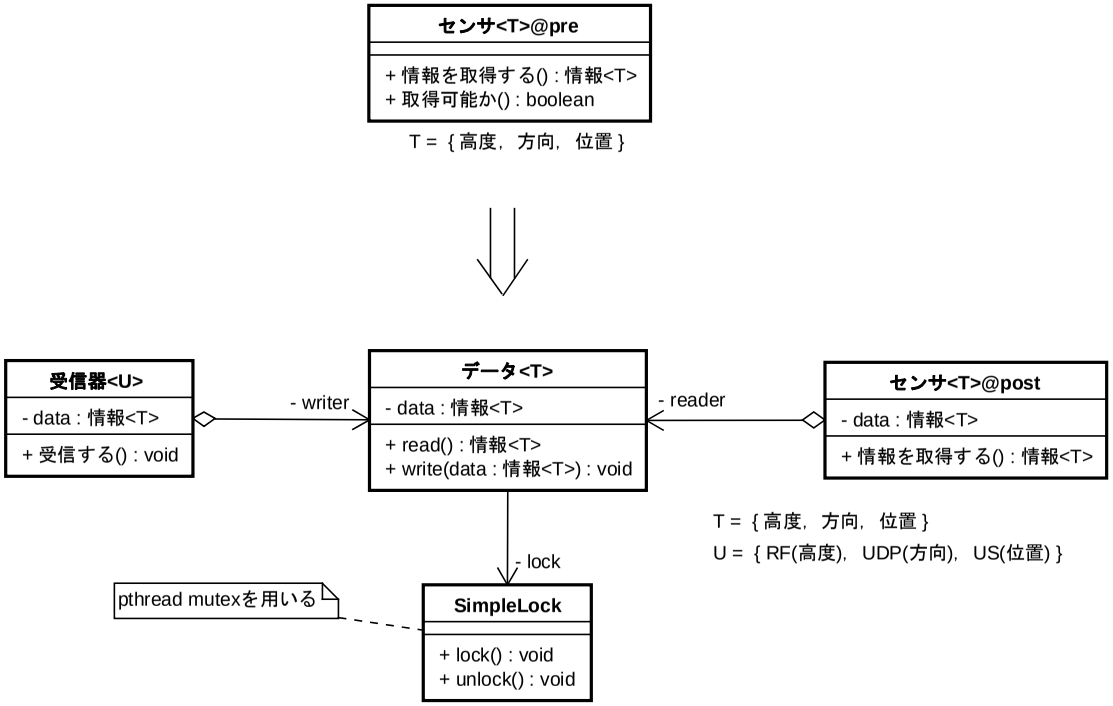

：飛行モード
：予測飛行
power := モータの出力を計算する(pos, dir, gpos)
altdiff := 高度差を求める(pos.z, gpos.z)
power := 出力を求める(altdiff)

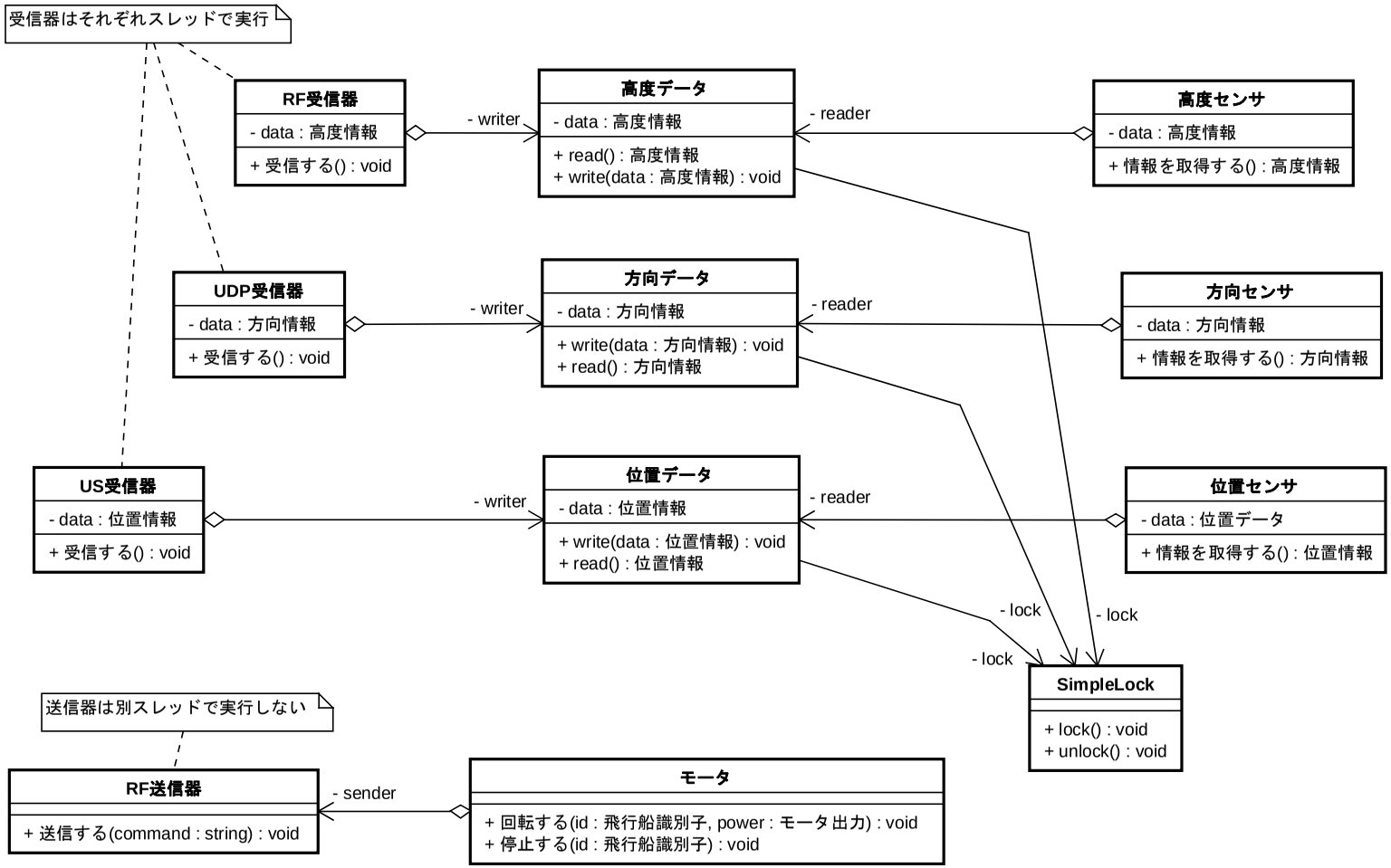

：飛行モード
：制御不能
power := モータの出力を計算する(pos, dir, gpos)
altdiff := 高度差を求める(pos.z, gpos.z)
power := 出力を求める(altdiff)

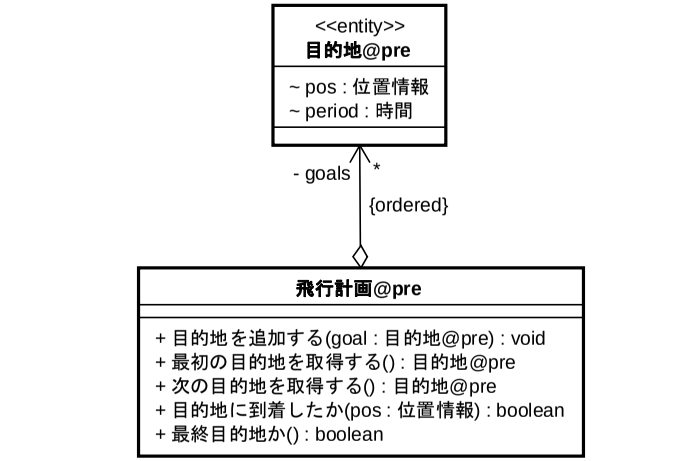

飛行可能
飛行特性とUSセンサ位置の設定済み
entry / status := READY_CHUS
飛行特性を設定する
USセンサの位置を設定する
USセンサの位置を設定する
飛行経路を設定する
飛行特性の設定済み
entry / status := READY_CH
飛行特性を設定する
飛行経路の設定済み
entry / status := READY_ALL
飛行経路を設定する
飛行特性を設定する
テスト飛行を開始する
USセンサ位置の設定済み
entry / status := READY_US
USセンサの位置を設定する
H
飛行特性を設定する
テストを終了する
自動操縦を開始する
着陸
USセンサの位置を設定する
テスト飛行を開始する
テスト飛行を開始する
初期
entry / status := INITIAL
自動飛行中
entry / status := PLAYING
システム停止
テスト飛行中
entry / status := TESTING

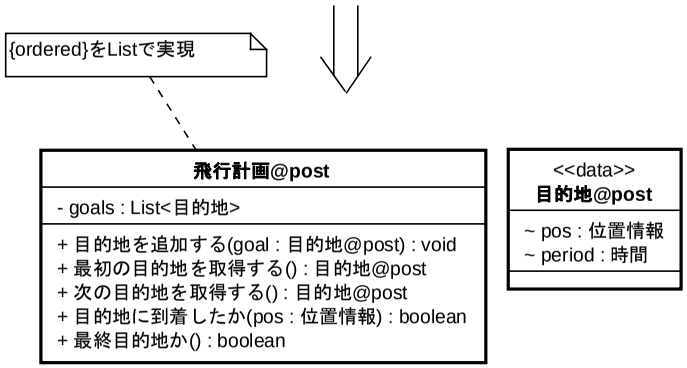

予測飛行
entry / mode = 予測飛行インスタンス
位置情報の取得に成功 / failures = 0
通常
entry / mode = 通常インスタンス
位置情報の取得に失敗 [failures < 失敗許容数] / failures++
位置情報が取得失敗 [failures = 失敗許容数]
自動飛行を開始する
位置情報の取得成功 / failures = 0
待機
entry / mode = NULL
制御不能
entry / mode = 制御不能インスタンス
システム停止

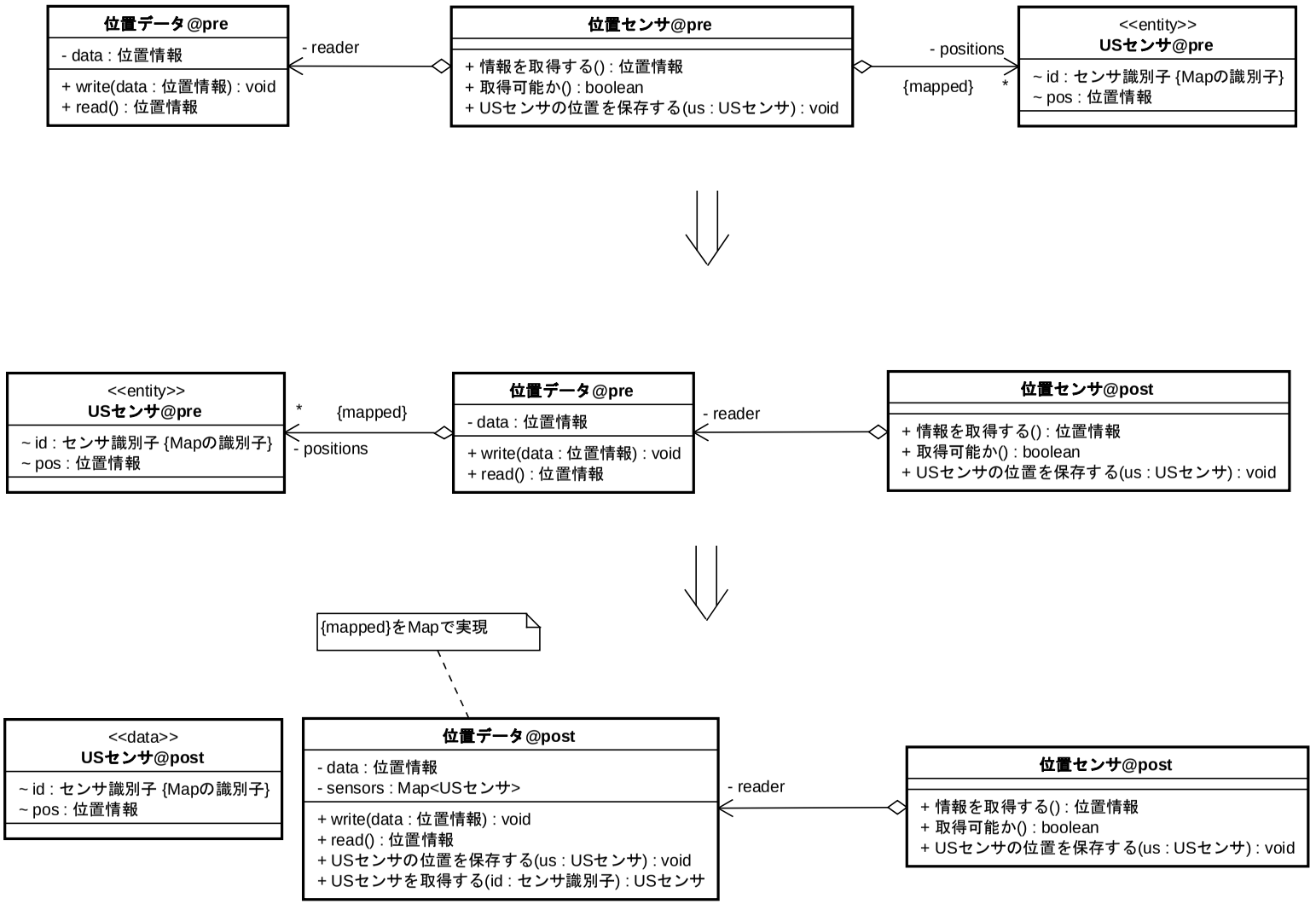

[予測時間が経過した] / passed := true
停止
カウント中
タイマ開始 / passes := false


センサ<T>@pre
+ 情報を取得する() : 情報<T>
+ 取得可能か() : boolean
T = { 高度，方向，位置 }
受信器<U>
- data : 情報<T>
+ 受信する() : void
- writer
データ<T>
- data : 情報<T>
+ read() : 情報<T>
+ write(data : 情報<T>) : void
- reader
センサ<T>@post
- data : 情報<T>
+ 情報を取得する() : 情報<T>
T = { 高度，方向，位置 }
U = { RF(高度)，UDP(方向)，US(位置) }
- lock
pthread mutexを用いる
SimpleLock
+ lock() : void
+ unlock() : void


受信器はそれぞれスレッドで実行
RF受信器
- data : 高度情報
+ 受信する() : void
- writer
高度データ
- data : 高度情報
+ read() : 高度情報
+ write(data : 高度情報) : void
- reader
高度センサ
- data : 高度情報
+ 情報を取得する() : 高度情報
UDP受信器
- data : 方向情報
+ 受信する() : void
- writer
方向データ
- data : 方向情報
+ write(data : 方向情報) : void
+ read() : 方向情報
- reader
方向センサ
- data : 方向情報
+ 情報を取得する() : 方向情報
US受信器
- data : 位置情報
+ 受信する() : void
- writer
位置データ
- data : 位置情報
+ write(data : 位置情報) : void
+ read() : 位置情報
- reader
位置センサ
- data : 位置データ
+ 情報を取得する() : 位置情報
- lock
- lock
- lock
SimpleLock
+ lock() : void
+ unlock() : void
送信器は別スレッドで実行しない
RF送信器
+ 送信する(command : string) : void
- sender
モータ
+ 回転する(id : 飛行船識別子, power : モータ出力) : void
+ 停止する(id : 飛行船識別子) : void


<<entity>>
目的地@pre
~ pos : 位置情報
~ period : 時間
- goals
*
{ordered}
飛行計画@pre
+ 目的地を追加する(goal : 目的地@pre) : void
+ 最初の目的地を取得する() : 目的地@pre
+ 次の目的地を取得する() : 目的地@pre
+ 目的地に到着したか(pos : 位置情報) : boolean
+ 最終目的地か() : boolean

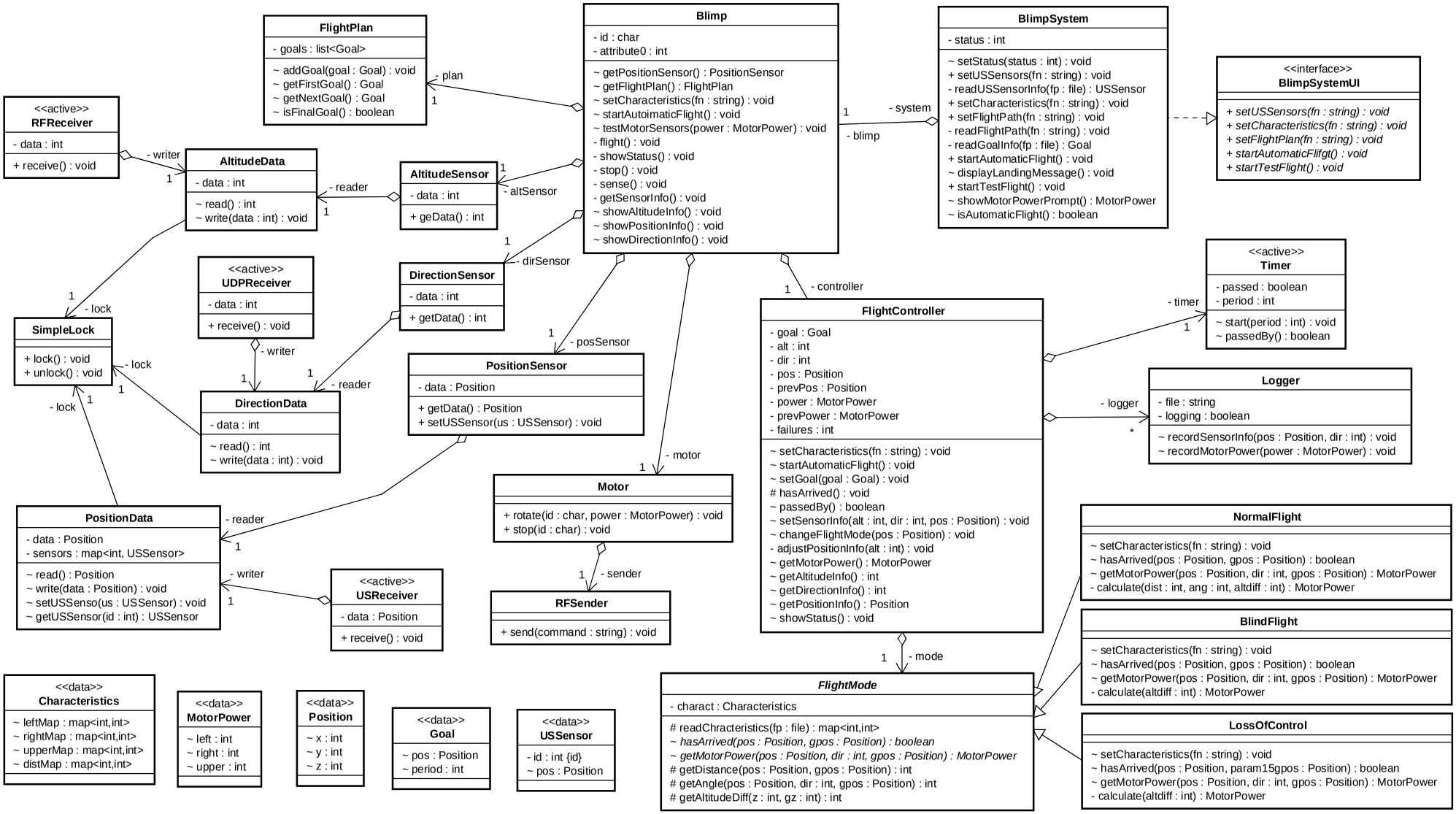

{ordered}をListで実現
飛行計画@post
- goals : List<目的地>
+ 目的地を追加する(goal : 目的地@post) : void
+ 最初の目的地を取得する() : 目的地@post
+ 次の目的地を取得する() : 目的地@post
+ 目的地に到着したか(pos : 位置情報) : boolean
+ 最終目的地か() : boolean
<<data>>
目的地@post
~ pos : 位置情報
~ period : 時間

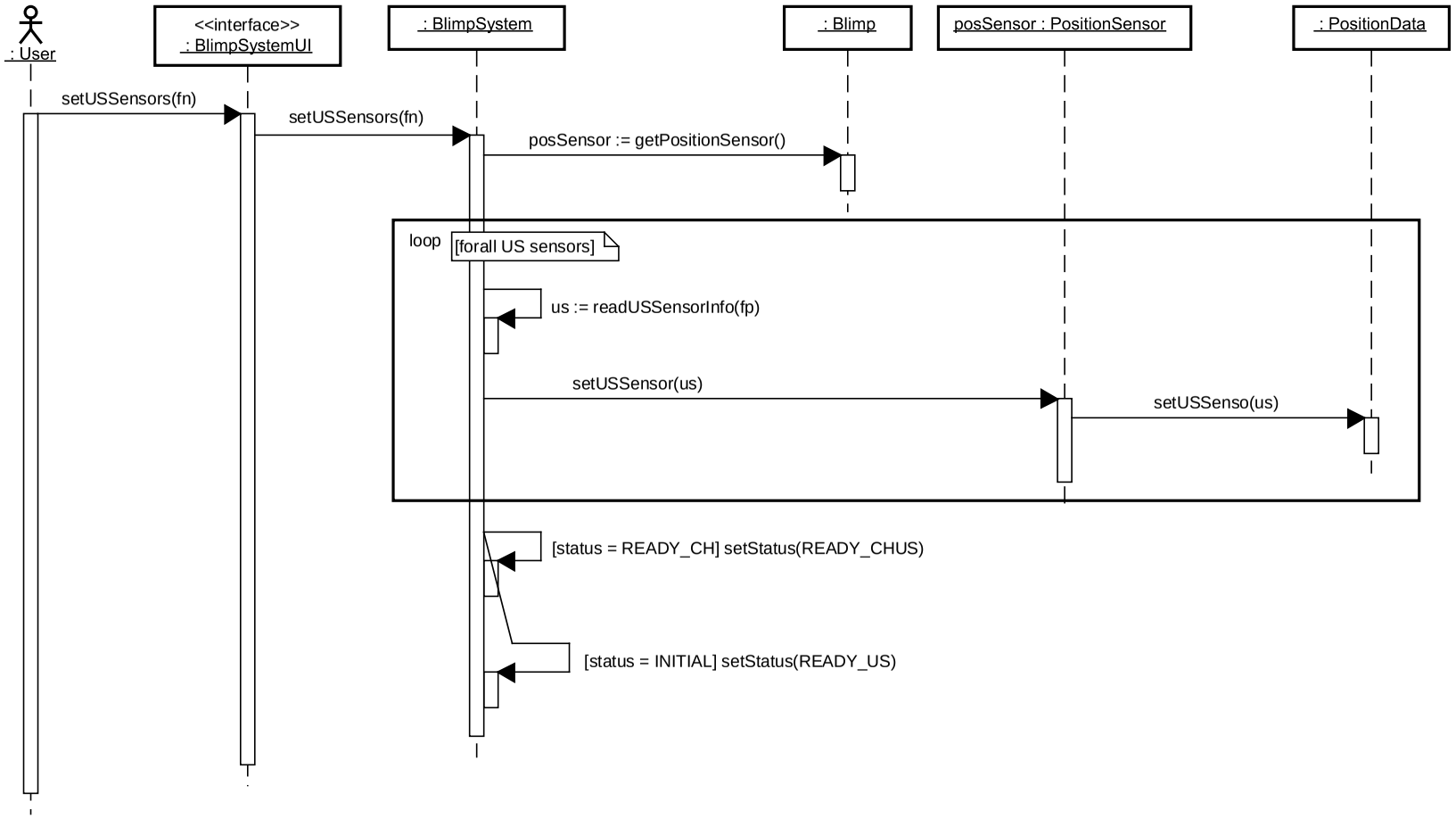

位置データ@pre
- data : 位置情報
+ write(data : 位置情報) : void
+ read() : 位置情報
- reader
位置センサ@pre
+ 情報を取得する() : 位置情報
+ 取得可能か() : boolean
+ USセンサの位置を保存する(us : USセンサ) : void
- positions
{mapped}
*
<<entity>>
USセンサ@pre
~ id : センサ識別子 {Mapの識別子}
~ pos : 位置情報
<<entity>>
USセンサ@pre
~ id : センサ識別子 {Mapの識別子}
~ pos : 位置情報
*
{mapped}
- positions
位置データ@pre
- data : 位置情報
+ write(data : 位置情報) : void
+ read() : 位置情報
- reader
位置センサ@post
+ 情報を取得する() : 位置情報
+ 取得可能か() : boolean
+ USセンサの位置を保存する(us : USセンサ) : void
{mapped}をMapで実現
<<data>>
USセンサ@post
~ id : センサ識別子 {Mapの識別子}
~ pos : 位置情報
位置データ@post
- data : 位置情報
- sensors : Map<USセンサ>
+ write(data : 位置情報) : void
+ read() : 位置情報
+ USセンサの位置を保存する(us : USセンサ) : void
+ USセンサを取得する(id : センサ識別子) : USセンサ
- reader
位置センサ@post
+ 情報を取得する() : 位置情報
+ 取得可能か() : boolean
+ USセンサの位置を保存する(us : USセンサ) : void

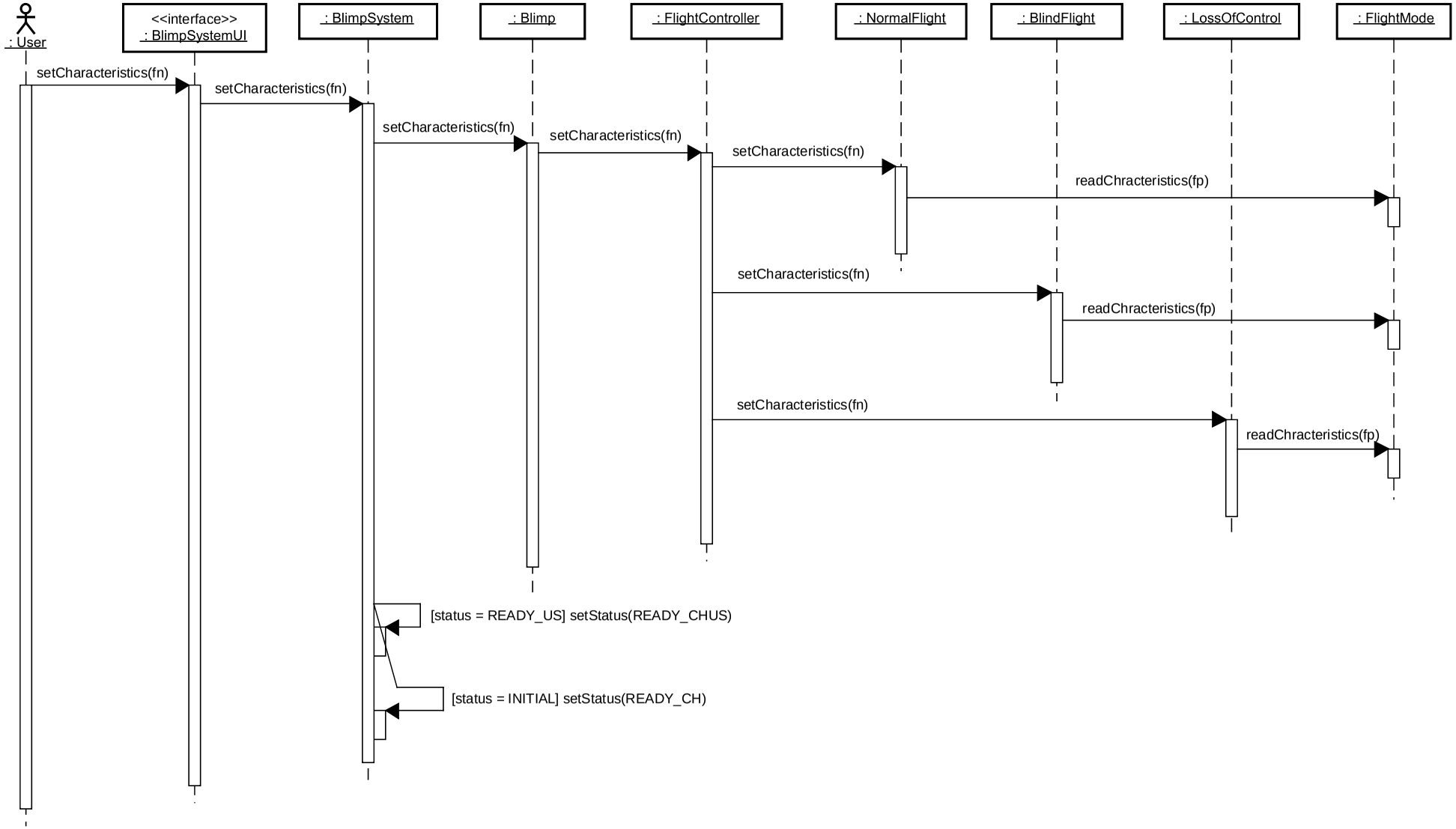
設計モデルにおけるクラス名と操作名の対応

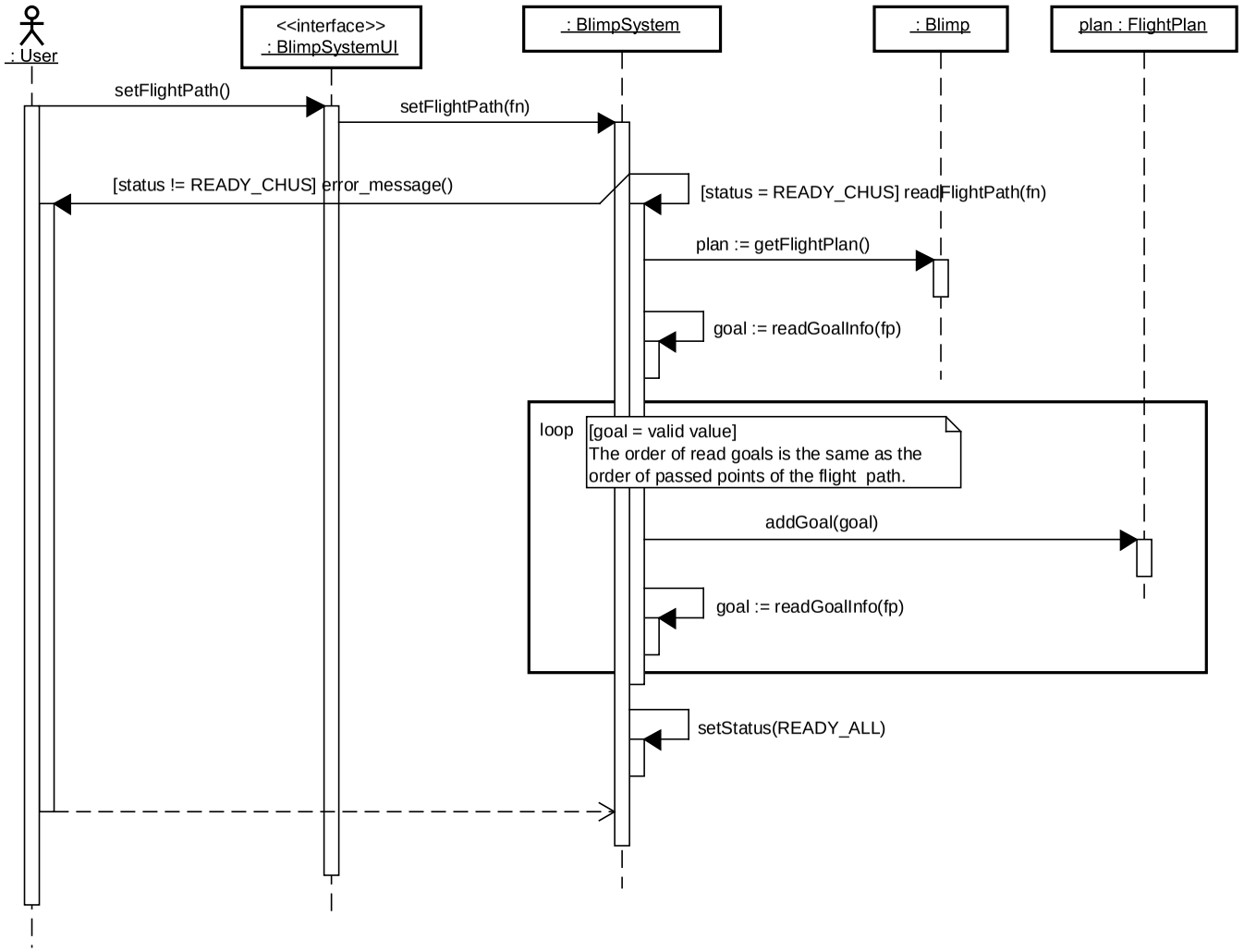
クラス名	
設計モデル(PIM)	設計モデル(PSM)
目的地	Goal
位置情報	Position
時間	int
座標値	int
飛行船システム	BlimpSystem
競技者UI	PlayerUI
保守者UI	MaintainerUI
システムの状態	int
飛行船	Blimp
飛行制御	FlightController
飛行モード	FlightMode
飛行計画	FlightPlan
位置センサ	PositionSensor
方向センサ	DirectionSensor
方向情報	int
高度センサ	AltitudeSensor
高度情報	int
USセンサ	USSensor
センサ識別子	int
モータ出力	MotorPower
モータ出力値	int
モータ	Motor
飛行船識別子	Characteristics
通常	NormalFlight
予測飛行	BlindFlight
制御不能	LossOfControl
タイマ	Timer
飛行特性	Characteristics
ロック器	ReadWriteLock
RF受信器	RFReceiver
UDP受信器	UDPReceiver
US受信器	USReceiver
RF送信器	RFSender
高度データ	AltitudeData
方向データ	DirectionData
位置データ	PositionData
ロック器	Locker

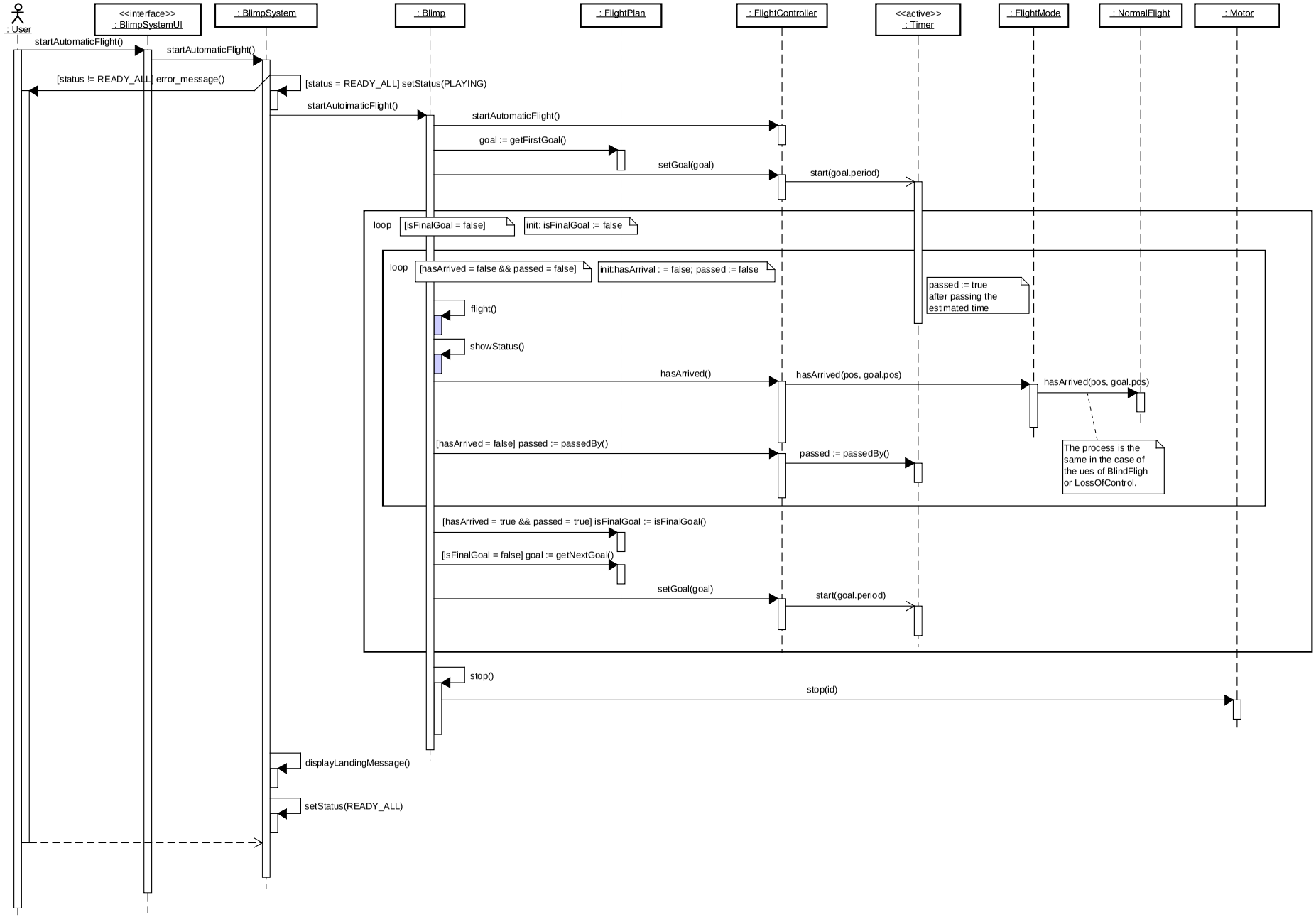
操作名	
設計モデル(PIM)	設計モデル(PSM)
USセンサの位置を設定する	setUSSensors
USセンサの位置を保存する	setUSSensor
USセンサの位置を読み込む	readUSSensorInfo
位置情報を取得する	getPositionInfo
位置情報を調整する	adjustPositionInfo
位置情報を表示する	showPositionInfo
位置センサを返す	getPositionSensor
回転する	rotate
角度を求める	getAngle
距離を求める	getDistance
高度差を求める	getAltitudeDiff
高度情報を取得する	getAltitudeInfo
高度情報を表示する	showAltitudeInfo
最終目的地か	isFinalGoal
最初の目的地を取得する	getFirstGoal
自動飛行中か	isAutomaticFlight
自動飛行を開始する	startAutomaticFlight
受信する	receive
出力を求める	calculate
状態を設定する	setStatus
情報を取得する	getData
センサ情報を取得する	getSensorInfo
センサ情報を設定する	setSensorInfo
センシングを行う	sense
送信する	send
タイマを開始する	start
着陸メッセージを表示する	displayLangingMessage
次の目的地を取得する	getNextGoal
停止する	stop
テスト飛行を開始する	startTestFlight
飛行計画を返す	getFlightPlan
飛行経路を設定する	setFlightPath
飛行経路を読み込む	readFlightPath
飛行状態を表示する	showStatus
飛行する	flight
飛行船の状況を表示する	showStatus
飛行特性を設定する	setCharacteristics
飛行特性を保存する	setCharacteristics
飛行特性を読み込む	readCharacteristics
飛行モードを切り替える	changeFlightMode
飛行を停止する	stop
方向情報を取得する	getDirectionInfo
方向情報を表示する	showDirectionInfo
モータ出力値の入力プロンプトを表示する	showMotorPowerPrompt
モータとセンサを試験する	testMotorSensors
モータの出力を計算する	getMotorPower
目的地に到着したか	hasArrived
目的地を設定する	setGoal
目的地を追加する	addGoal
目的地を読み込む	readGoalInfo
予測飛行時間が経過したか	passedBy


FlightPlan
- goals : list<Goal>
~ addGoal(goal : Goal) : void
~ getFirstGoal() : Goal
~ getNextGoal() : Goal
~ isFinalGoal() : boolean
- plan
1
Blimp
- id : char
- attribute0 : int
~ getPositionSensor() : PositionSensor
~ getFlightPlan() : FlightPlan
~ setCharacteristics(fn : string) : void
~ startAutoimaticFlight() : void
~ testMotorSensors(power : MotorPower) : void
- flight() : void
- showStatus() : void
- stop() : void
- sense() : void
- getSensorInfo() : void
~ showAltitudeInfo() : void
~ showPositionInfo() : void
~ showDirectionInfo() : void
1
- system
- blimp
BlimpSystem
- status : int
~ setStatus(status : int) : void
+ setUSSensors(fn : string) : void
- readUSSensorInfo(fp : file) : USSensor
+ setCharacteristics(fn : string) : void
+ setFlightPath(fn : string) : void
- readFlightPath(fn : string) : void
- readGoalInfo(fp : file) : Goal
+ startAutomaticFlight() : void
~ displayLandingMessage() : void
+ startTestFlight() : void
~ showMotorPowerPrompt() : MotorPower
~ isAutomaticFlight() : boolean
<<interface>>
BlimpSystemUI
+ setUSSensors(fn : string) : void
+ setCharacteristics(fn : string) : void
+ setFlightPlan(fn : string) : void
+ startAutomaticFlifgt() : void
+ startTestFlight() : void
<<active>>
RFReceiver
- data : int
+ receive() : void
- writer
1
AltitudeData
- data : int
~ read() : int
~ write(data : int) : void
- reader
1
AltitudeSensor
- data : int
+ geData() : int
1
- altSensor
1
- dirSensor
<<active>>
UDPReceiver
- data : int
+ receive() : void
DirectionSensor
- data : int
+ getData() : int
1
- lock
SimpleLock
+ lock() : void
+ unlock() : void
- lock
1
- lock
1
- writer
1
DirectionData
- data : int
~ read() : int
~ write(data : int) : void
1
- reader
1
- posSensor
PositionSensor
- data : Position
+ getData() : Position
+ setUSSensor(us : USSensor) : void
1
- motor
Motor
+ rotate(id : char, power : MotorPower) : void
+ stop(id : char) : void
1
- sender
RFSender
+ send(command : string) : void
1
- controller
FlightController
- goal : Goal
- alt : int
- dir : int
- pos : Position
- prevPos : Position
- power : MotorPower
- prevPower : MotorPower
- failures : int
~ setCharacteristics(fn : string) : void
~ startAutomaticFlight() : void
~ setGoal(goal : Goal) : void
# hasArrived() : void
~ passedBy() : boolean
~ setSensorInfo(alt : int, dir : int, pos : Position) : void
~ changeFlightMode(pos : Position) : void
- adjustPositionInfo(alt : int) : void
~ getMotorPower() : MotorPower
~ getAltitudeInfo() : int
~ getDirectionInfo() : int
~ getPositionInfo() : Position
~ showStatus() : void
- timer
1
<<active>>
Timer
- passed : boolean
- period : int
~ start(period : int) : void
~ passedBy() : boolean
- logger
*
Logger
- file : string
- logging : boolean
~ recordSensorInfo(pos : Position, dir : int) : void
~ recordMotorPower(power : MotorPower) : void
PositionData
- data : Position
- sensors : map<int, USSensor>
~ read() : Position
~ write(data : Position) : void
~ setUSSenso(us : USSensor) : void
~ getUSSensor(id : int) : USSensor
- reader
1
- writer
1
<<active>>
USReceiver
- data : Position
+ receive() : void
1
- mode
FlightMode
- charact : Characteristics
readChracteristics(fp : file) : map<int,int>
~ hasArrived(pos : Position, gpos : Position) : boolean
~ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
# getDistance(pos : Position, gpos : Position) : int
# getAngle(pos : Position, dir : int, gpos : Position) : int
# getAltitudeDiff(z : int, gz : int) : int
NormalFlight
~ setCharacteristics(fn : string) : void
~ hasArrived(pos : Position, gpos : Position) : boolean
~ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
- calculate(dist : int, ang : int, altdiff : int) : MotorPower
BlindFlight
~ setCharacteristics(fn : string) : void
~ hasArrived(pos : Position, gpos : Position) : boolean
~ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
- calculate(altdiff : int) : MotorPower
LossOfControl
~ setCharacteristics(fn : string) : void
~ hasArrived(pos : Position, param15gpos : Position) : boolean
~ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
- calculate(altdiff : int) : MotorPower
<<data>>
Characteristics
~ leftMap : map<int,int>
~ rightMap : map<int,int>
~ upperMap : map<int,int>
~ distMap : map<int,int>
<<data>>
MotorPower
~ left : int
~ right : int
~ upper : int
<<data>>
Position
~ x : int
~ y : int
~ z : int
<<data>>
Goal
~ pos : Position
~ period : int
<<data>>
USSensor
- id : int {id}
~ pos : Position

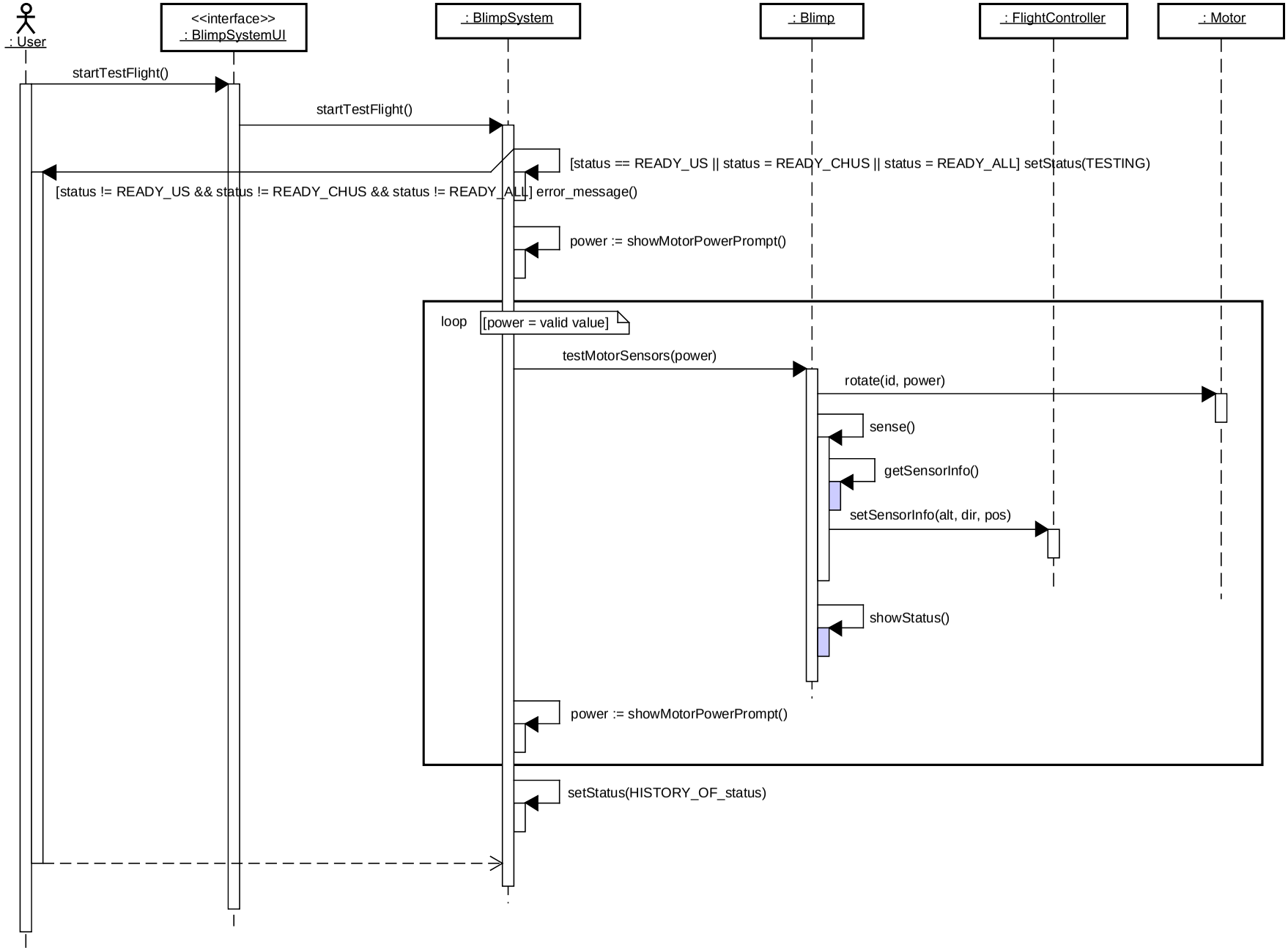

: User
<<interface>>
: BlimpSystemUI
: BlimpSystem
: Blimp
posSensor : PositionSensor
: PositionData
setUSSensors(fn)
setUSSensors(fn)
posSensor := getPositionSensor()
loop
[forall US sensors]
us := readUSSensorInfo(fp)
setUSSensor(us)
setUSSenso(us)
[status = READY_CH] setStatus(READY_CHUS)
[status = INITIAL] setStatus(READY_US)

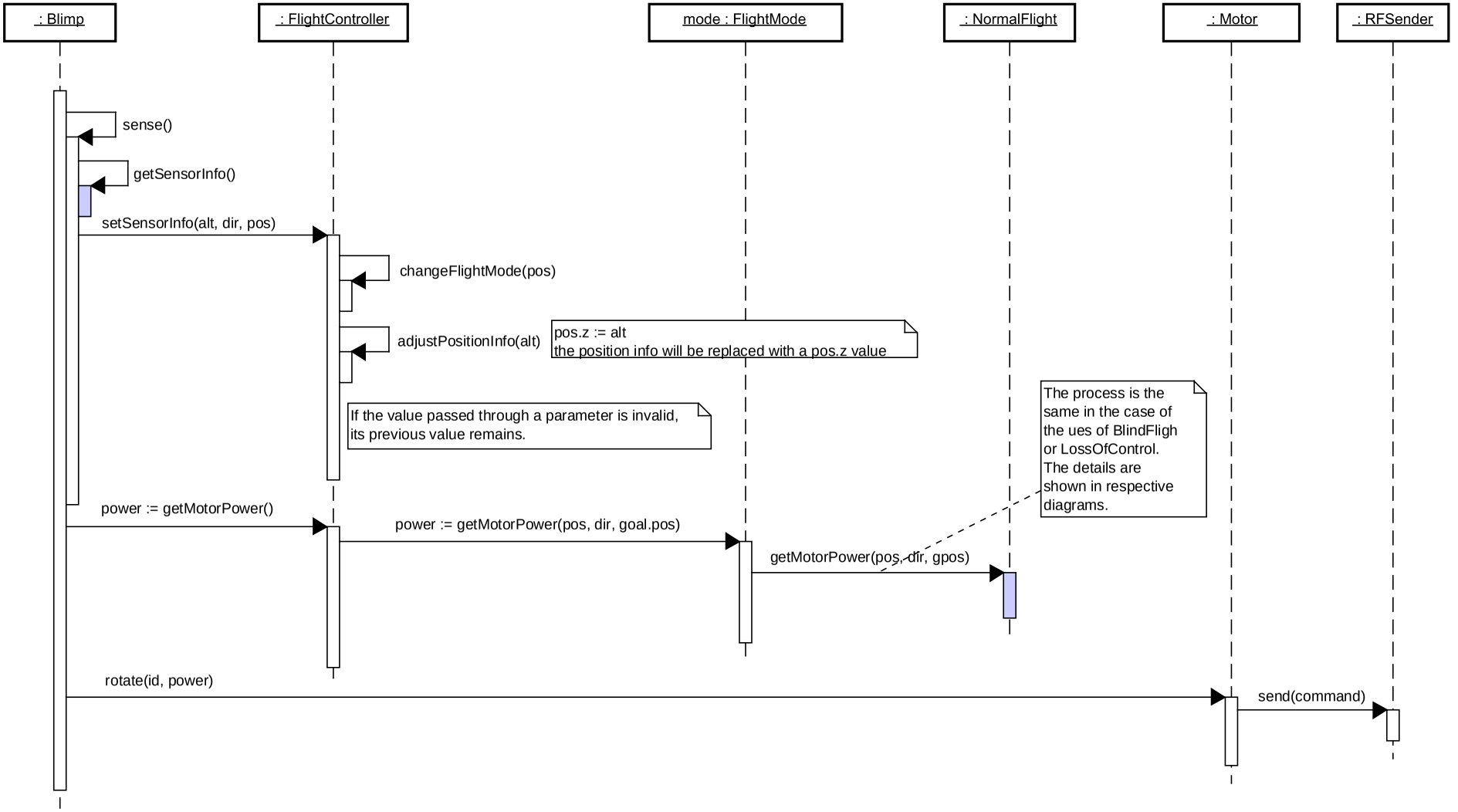

: User
<<interface>>
: BlimpSystemUI
: BlimpSystem
: Blimp
: FlightController
: NormalFlight
: BlindFlight
: LossOfControl
: FlightMode
setCharacteristics(fn)
setCharacteristics(fn)
setCharacteristics(fn)
setCharacteristics(fn)
setCharacteristics(fn)
readChracteristics(fp)
setCharacteristics(fn)
readChracteristics(fp)
setCharacteristics(fn)
readChracteristics(fp)
[status = READY_US] setStatus(READY_CHUS)
[status = INITIAL] setStatus(READY_CH)

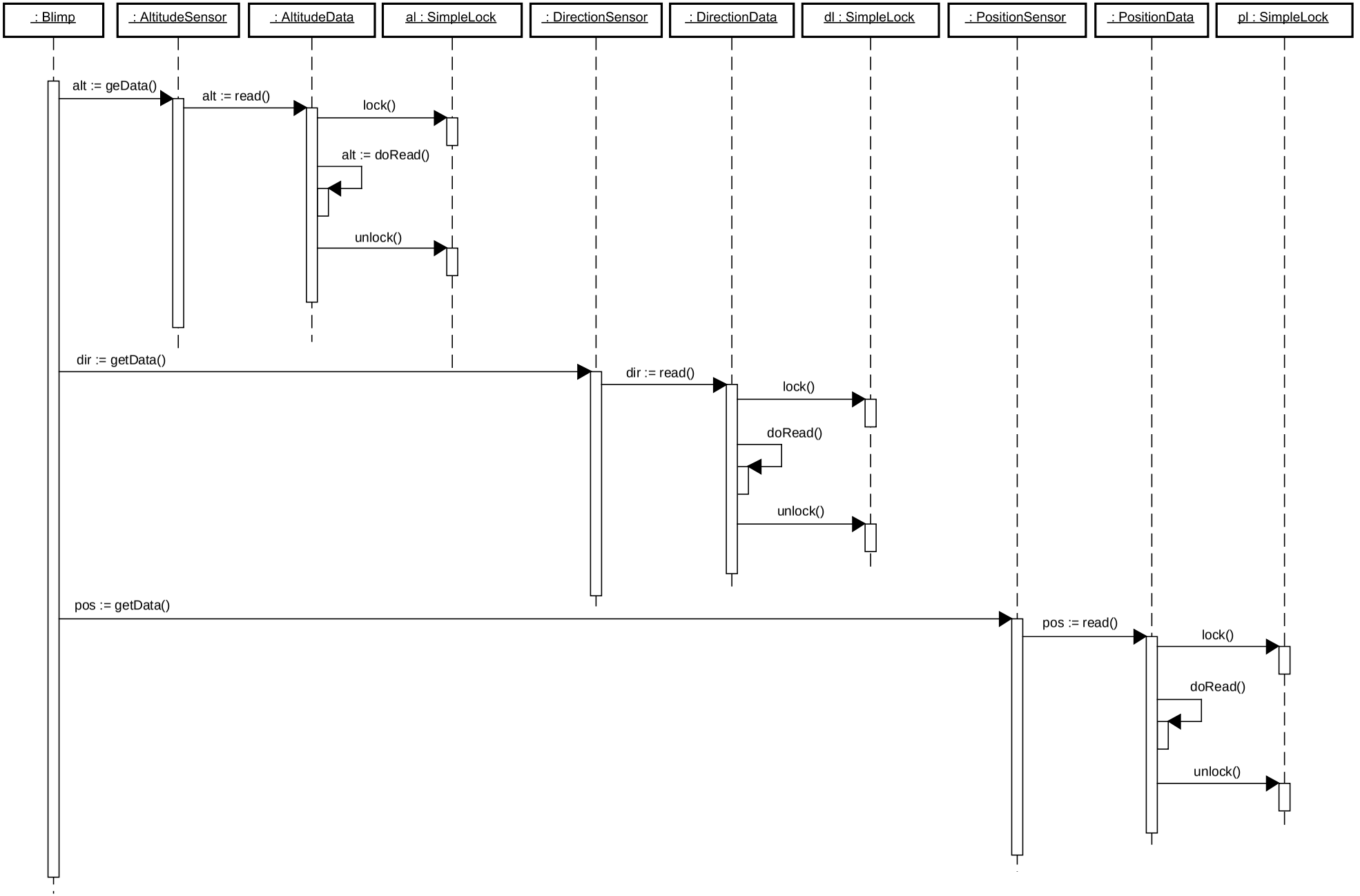

: User
<<interface>>
: BlimpSystemUI
: BlimpSystem
: Blimp
plan : FlightPlan
setFlightPath()
setFlightPath(fn)
[status != READY_CHUS] error_message()
[status = READY_CHUS] readFlightPath(fn)
plan := getFlightPlan()
goal := readGoalInfo(fp)
loop
[goal = valid value]
The order of read goals is the same as the order of passed points of the flight path.
addGoal(goal)
goal := readGoalInfo(fp)
setStatus(READY_ALL)

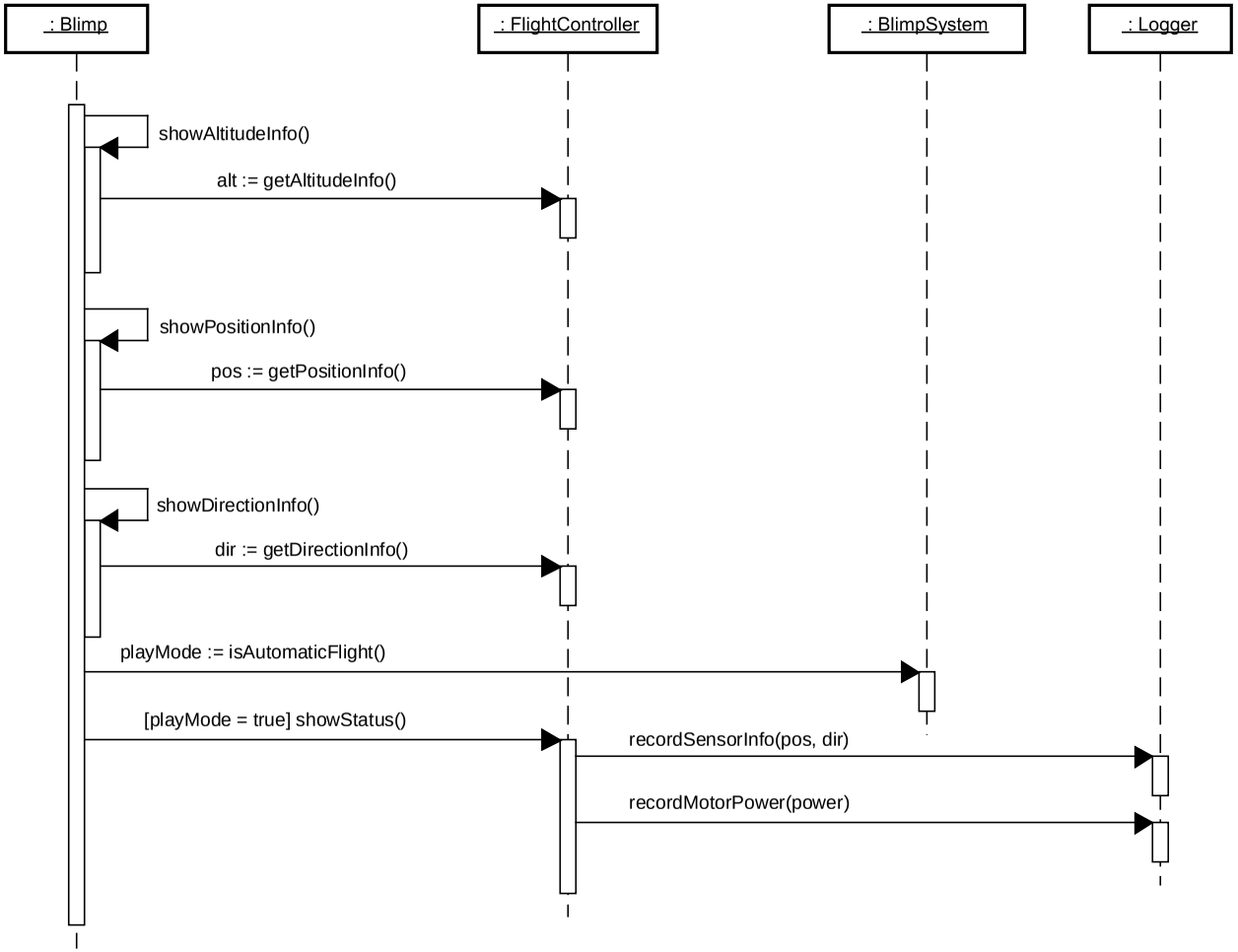

: User
<<interface>>
: BlimpSystemUI
: BlimpSystem
: Blimp
: FlightPlan
: FlightController
<<active>>
: Timer
: FlightMode
: NormalFlight
: Motor
startAutomaticFlight()
startAutomaticFlight()
[status != READY_ALL] error_message()
[status = READY_ALL] setStatus(PLAYING)
startAutoimaticFlight()
startAutomaticFlight()
goal := getFirstGoal()
setGoal(goal)
start(goal.period)
loop
[isFinalGoal = false]
init: isFinalGoal := false
loop
[hasArrived = false && passed = false]
init:hasArrival : = false; passed := false
passed := true
after passing the
estimated time
flight()
showStatus()
hasArrived()
hasArrived(pos, goal.pos)
hasArrived(pos, goal.pos)
[hasArrived = false] passed := passedBy()
passed := passedBy()
The process is the
same in the case of
the ues of BlindFligh
or LossOfControl.
[hasArrived = true && passed = true] isFinalGoal := isFinalGoal()
[isFinalGoal = false] goal := getNextGoal()
setGoal(goal)
start(goal.period)
stop()
stop(id)
displayLandingMessage()
setStatus(READY_ALL)

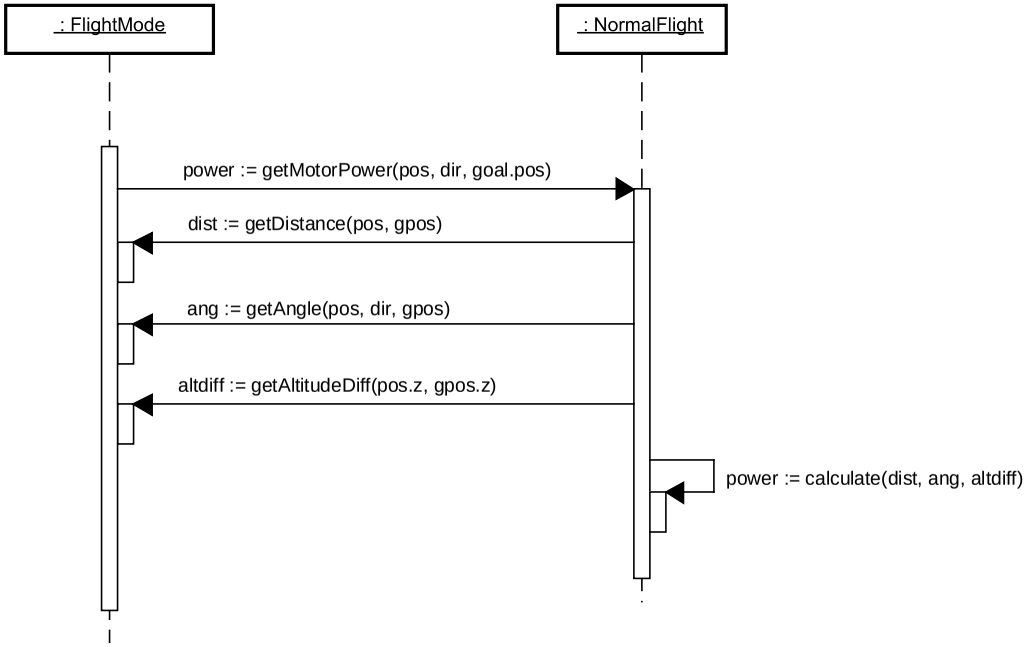
詳細設計(5/13)

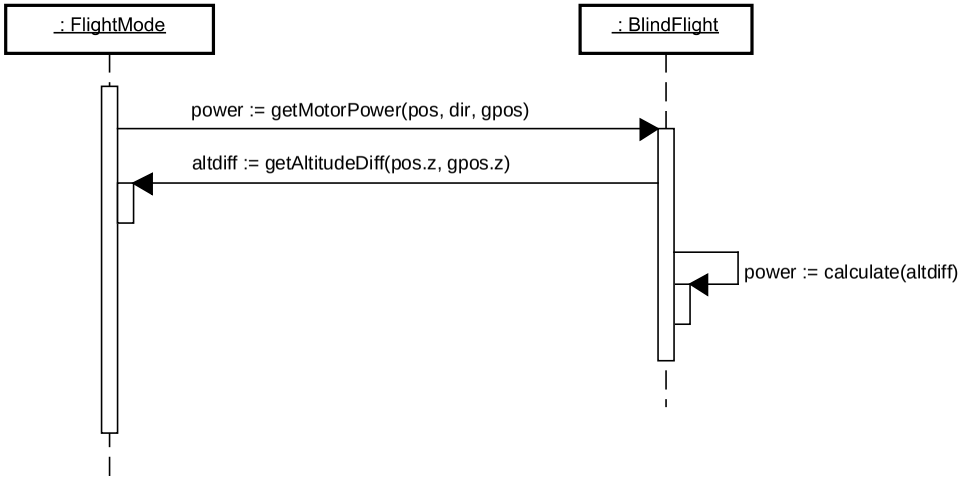

: User
<<interface>>
: BlimpSystemUI
: BlimpSystem
: Blimp
: FlightController
: Motor
startTestFlight()
startTestFlight()
[status == READY_US || status = READY_CHUS || status = READY_ALL] setStatus(TESTING)
[status != READY_US && status != READY_CHUS && status != READY_ALL] error_message()
power := showMotorPowerPrompt()
loop
[power = valid value]
testMotorSensors(power)
rotate(id, power)
sense()
getSensorInfo()
setSensorInfo(alt, dir, pos)
showStatus()
power := showMotorPowerPrompt()
setStatus(HISTORY_OF_status)

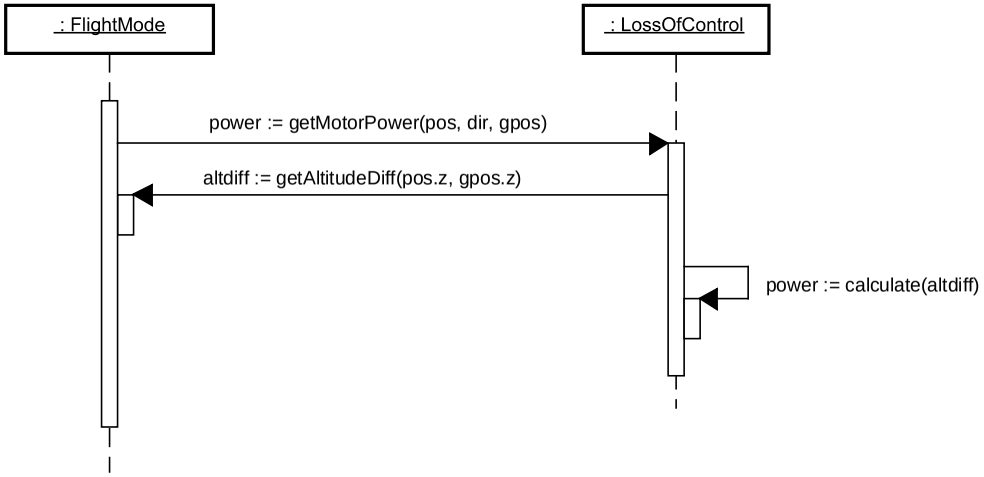

: Blimp
: FlightController
mode : FlightMode
: NormalFlight
: Motor
: RFSender
sense()
getSensorInfo()
setSensorInfo(alt, dir, pos)
changeFlightMode(pos)
adjustPositionInfo(alt)
pos.z := alt
the position info will be replaced with a pos.z value
If the value passed through a parameter is invalid,
its previous value remains.
The process is the same in the case of the ues of BlindFligh or LossOfControl. The details are shown in respective diagrams.
power := getMotorPower()
power := getMotorPower(pos, dir, goal.pos)
getMotorPower(pos, dir, gpos)
rotate(id, power)
send(command)

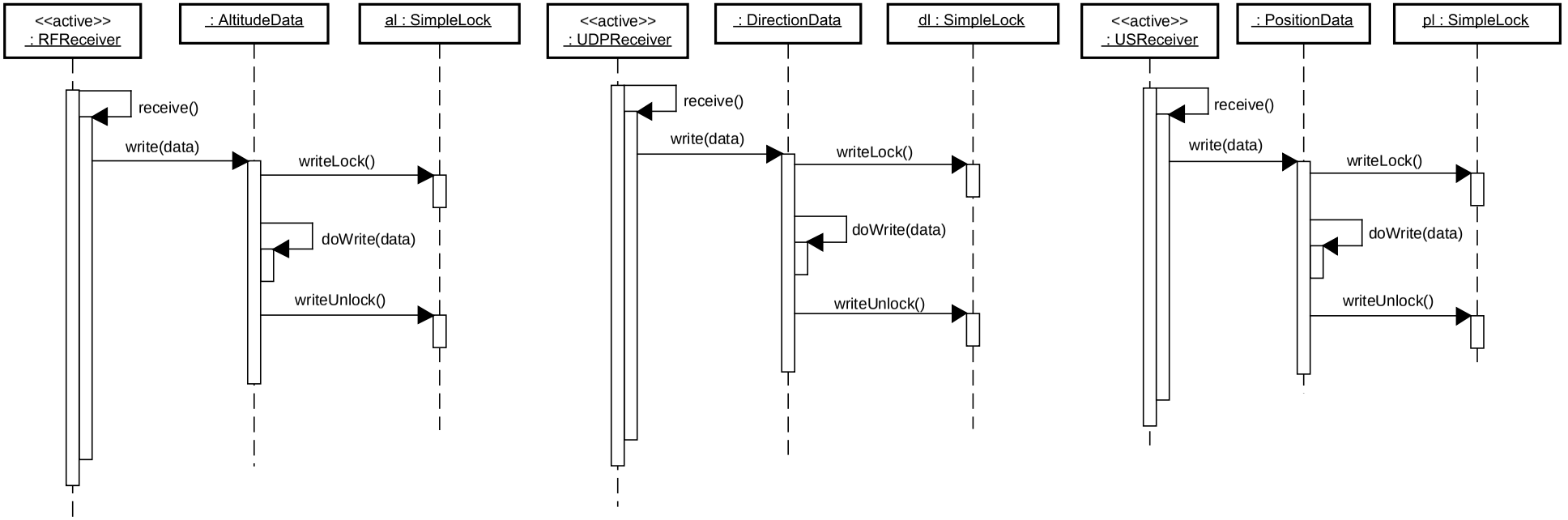

: Blimp
: AltitudeSensor
: AltitudeData
al : SimpleLock
: DirectionSensor
: DirectionData
dl : SimpleLock
: PositionSensor
: PositionData
pl : SimpleLock
alt := geData()
alt := read()
lock()
alt := doRead()
unlock()
dir := getData()
dir := read()
lock()
doRead()
unlock()
pos := getData()
pos := read()
lock()
doRead()
unlock()

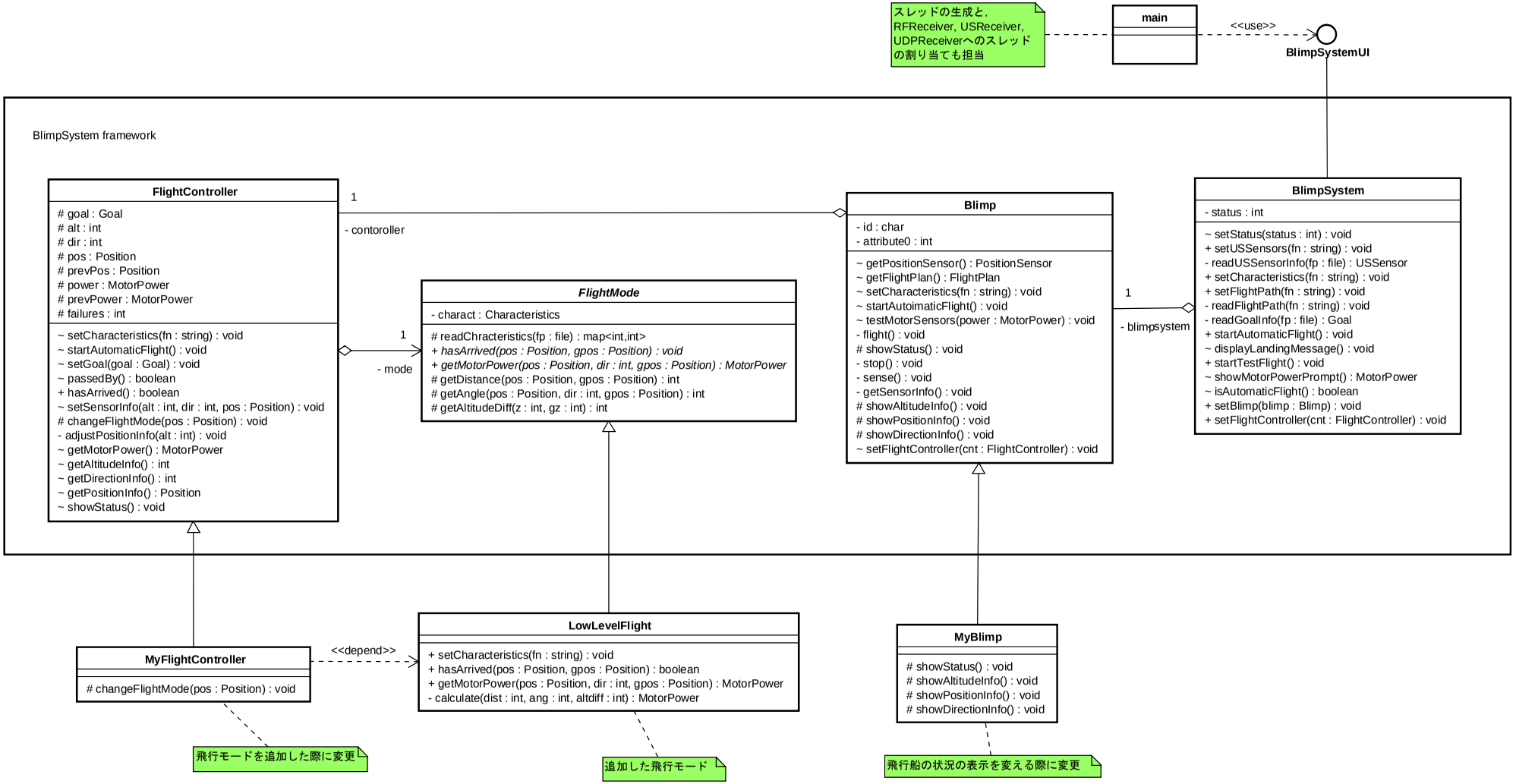

: Blimp
: FlightController
: BlimpSystem
: Logger
showAltitudeInfo()
alt := getAltitudeInfo()
showPositionInfo()
pos := getPositionInfo()
showDirectionInfo()
dir := getDirectionInfo()
playMode := isAutomaticFlight()
[playMode = true] showStatus()
recordSensorInfo(pos, dir)
recordMotorPower(power)

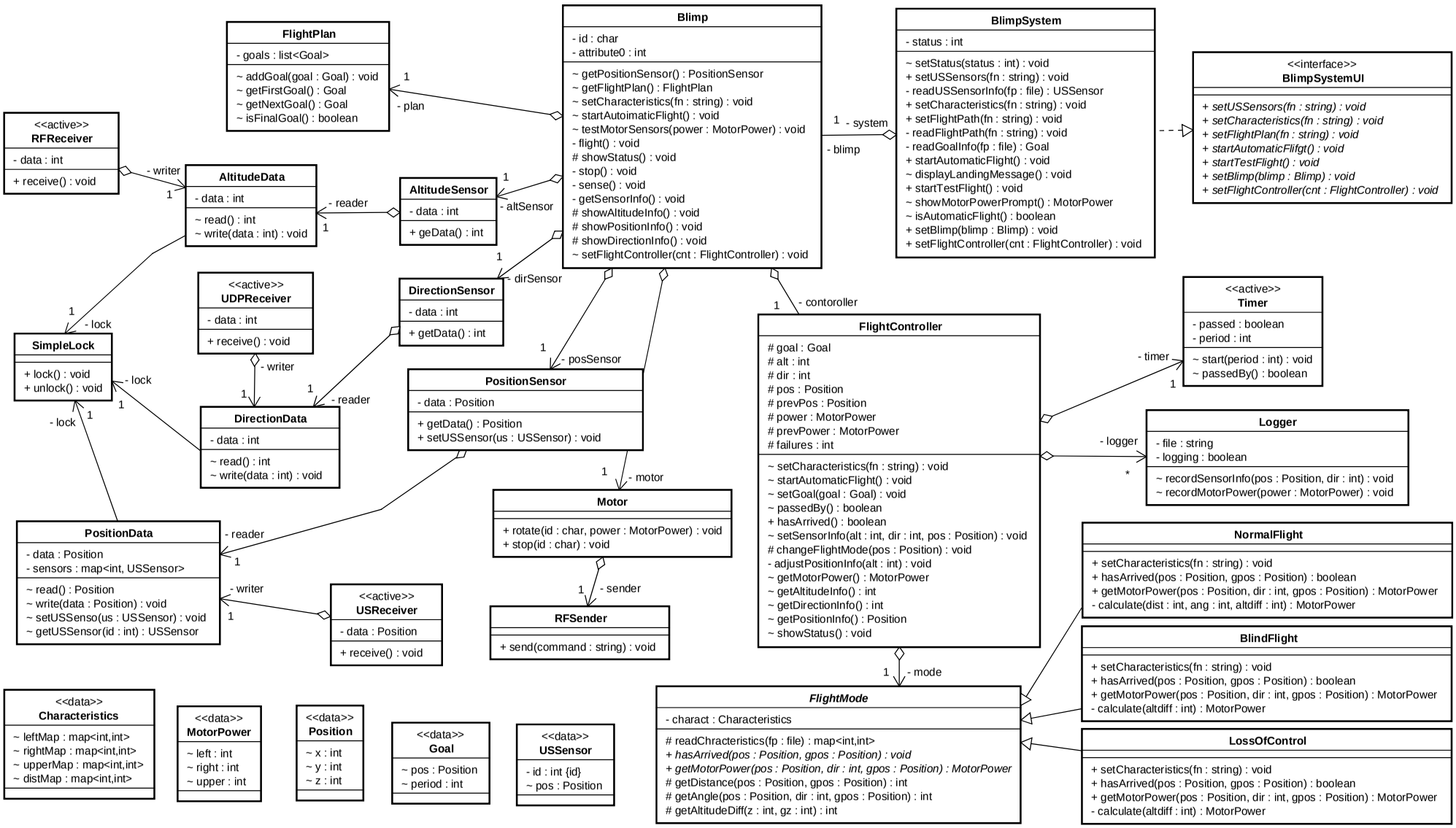

: FlightMode
: NormalFlight
power := getMotorPower(pos, dir, goal.pos)
dist := getDistance(pos, gpos)
ang := getAngle(pos, dir, gpos)
altdiff := getAltitudeDiff(pos.z, gpos.z)
power := calculate(dist, ang, altdiff)


: FlightMode
: BlindFlight
power := getMotorPower(pos, dir, gpos)
altdiff := getAltitudeDiff(pos.z, gpos.z)
power := calculate(altdiff)


: FlightMode
: LossOfControl
power := getMotorPower(pos, dir, gpos)
altdiff := getAltitudeDiff(pos.z, gpos.z)
power := calculate(altdiff)


<<active>>
: RFReceiver
: AltitudeData
al : SimpleLock
receive()
write(data)
writeLock()
doWrite(data)
writeUnlock()
<<active>>
: UDPReceiver
: DirectionData
dl : SimpleLock
receive()
write(data)
writeLock()
doWrite(data)
writeUnlock()
<<active>>
: USReceiver
: PositionData
pl : SimpleLock
receive()
write(data)
writeLock()
doWrite(data)
writeUnlock()


スレッドの生成と,
RFReceiver, USReceiver,
UDPReceiverへのスレッド
の割り当ても担当
main
<<use>>
BlimpSystemUI
BlimpSystem framework
FlightController
goal : Goal
alt : int
dir : int
pos : Position
prevPos : Position
power : MotorPower
prevPower : MotorPower
failures : int
~ setCharacteristics(fn : string) : void
~ startAutomaticFlight() : void
~ setGoal(goal : Goal) : void
~ passedBy() : boolean
+ hasArrived() : boolean
~ setSensorInfo(alt : int, dir : int, pos : Position) : void
# changeFlightMode(pos : Position) : void
- adjustPositionInfo(alt : int) : void
~ getMotorPower() : MotorPower
~ getAltitudeInfo() : int
~ getDirectionInfo() : int
~ getPositionInfo() : Position
~ showStatus() : void
1
- contoroller
1
- mode
FlightMode
- charact : Characteristics
readChracteristics(fp : file) : map<int,int>
+ hasArrived(pos : Position, gpos : Position) : void
+ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
# getDistance(pos : Position, gpos : Position) : int
# getAngle(pos : Position, dir : int, gpos : Position) : int
# getAltitudeDiff(z : int, gz : int) : int
Blimp
- id : char
- attribute0 : int
~ getPositionSensor() : PositionSensor
~ getFlightPlan() : FlightPlan
~ setCharacteristics(fn : string) : void
~ startAutoimaticFlight() : void
~ testMotorSensors(power : MotorPower) : void
- flight() : void
showStatus() : void
- stop() : void
- sense() : void
- getSensorInfo() : void
showAltitudeInfo() : void
showPositionInfo() : void
showDirectionInfo() : void
~ setFlightController(cnt : FlightController) : void
1
- blimpsystem
BlimpSystem
- status : int
~ setStatus(status : int) : void
+ setUSSensors(fn : string) : void
- readUSSensorInfo(fp : file) : USSensor
+ setCharacteristics(fn : string) : void
+ setFlightPath(fn : string) : void
- readFlightPath(fn : string) : void
- readGoalInfo(fp : file) : Goal
+ startAutomaticFlight() : void
~ displayLandingMessage() : void
+ startTestFlight() : void
~ showMotorPowerPrompt() : MotorPower
~ isAutomaticFlight() : boolean
+ setBlimp(blimp : Blimp) : void
+ setFlightController(cnt : FlightController) : void
MyFlightController
changeFlightMode(pos : Position) : void
<<depend>>
LowLevelFlight
+ setCharacteristics(fn : string) : void
+ hasArrived(pos : Position, gpos : Position) : boolean
+ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
- calculate(dist : int, ang : int, altdiff : int) : MotorPower
MyBlimp
showStatus() : void
showAltitudeInfo() : void
showPositionInfo() : void
showDirectionInfo() : void
飛行モードを追加した際に変更
追加した飛行モード
飛行船の状況の表示を変える際に変更


FlightPlan
- goals : list<Goal>
~ addGoal(goal : Goal) : void
~ getFirstGoal() : Goal
~ getNextGoal() : Goal
~ isFinalGoal() : boolean
Blimp
- id : char
- attribute0 : int
~ getPositionSensor() : PositionSensor
~ getFlightPlan() : FlightPlan
~ setCharacteristics(fn : string) : void
~ startAutoimaticFlight() : void
~ testMotorSensors(power : MotorPower) : void
- flight() : void
showStatus() : void
- stop() : void
- sense() : void
- getSensorInfo() : void
showAltitudeInfo() : void
showPositionInfo() : void
showDirectionInfo() : void
~ setFlightController(cnt : FlightController) : void
BlimpSystem
- status : int
~ setStatus(status : int) : void
+ setUSSensors(fn : string) : void
- readUSSensorInfo(fp : file) : USSensor
+ setCharacteristics(fn : string) : void
+ setFlightPath(fn : string) : void
- readFlightPath(fn : string) : void
- readGoalInfo(fp : file) : Goal
+ startAutomaticFlight() : void
~ displayLandingMessage() : void
+ startTestFlight() : void
~ showMotorPowerPrompt() : MotorPower
~ isAutomaticFlight() : boolean
+ setBlimp(blimp : Blimp) : void
+ setFlightController(cnt : FlightController) : void
<<interface>>
BlimpSystemUI
+ setUSSensors(fn : string) : void
+ setCharacteristics(fn : string) : void
+ setFlightPlan(fn : string) : void
+ startAutomaticFlifgt() : void
+ startTestFlight() : void
+ setBlimp(blimp : Blimp) : void
+ setFlightController(cnt : FlightController) : void
<<active>>
RFReceiver
- data : int
+ receive() : void
AltitudeData
- data : int
~ read() : int
~ write(data : int) : void
AltitudeSensor
- data : int
+ geData() : int
<<active>>
UDPReceiver
- data : int
+ receive() : void
DirectionSensor
- data : int
+ getData() : int
SimpleLock
+ lock() : void
+ unlock() : void
DirectionData
- data : int
~ read() : int
~ write(data : int) : void
PositionSensor
- data : Position
+ getData() : Position
+ setUSSensor(us : USSensor) : void
FlightController
goal : Goal
alt : int
dir : int
pos : Position
prevPos : Position
power : MotorPower
prevPower : MotorPower
failures : int
~ setCharacteristics(fn : string) : void
~ startAutomaticFlight() : void
~ setGoal(goal : Goal) : void
~ passedBy() : boolean
+ hasArrived() : boolean
~ setSensorInfo(alt : int, dir : int, pos : Position) : void
# changeFlightMode(pos : Position) : void
- adjustPositionInfo(alt : int) : void
~ getMotorPower() : MotorPower
~ getAltitudeInfo() : int
~ getDirectionInfo() : int
~ getPositionInfo() : Position
~ showStatus() : void
<<active>>
Timer
- passed : boolean
- period : int
~ start(period : int) : void
~ passedBy() : boolean
Logger
- file : string
- logging : boolean
~ recordSensorInfo(pos : Position, dir : int) : void
~ recordMotorPower(power : MotorPower) : void
Motor
+ rotate(id : char, power : MotorPower) : void
+ stop(id : char) : void
PositionData
- data : Position
- sensors : map<int, USSensor>
~ read() : Position
~ write(data : Position) : void
~ setUSSenso(us : USSensor) : void
~ getUSSensor(id : int) : USSensor
<<active>>
USReceiver
- data : Position
+ receive() : void
RFSender
+ send(command : string) : void
NormalFlight
+ setCharacteristics(fn : string) : void
+ hasArrived(pos : Position, gpos : Position) : boolean
+ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
- calculate(dist : int, ang : int, altdiff : int) : MotorPower
BlindFlight
+ setCharacteristics(fn : string) : void
+ hasArrived(pos : Position, gpos : Position) : boolean
+ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
- calculate(altdiff : int) : MotorPower
LossOfControl
+ setCharacteristics(fn : string) : void
+ hasArrived(pos : Position, gpos : Position) : boolean
+ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
- calculate(altdiff : int) : MotorPower
FlightMode
- charact : Characteristics
readChracteristics(fp : file) : map<int,int>
+ hasArrived(pos : Position, gpos : Position) : void
+ getMotorPower(pos : Position, dir : int, gpos : Position) : MotorPower
# getDistance(pos : Position, gpos : Position) : int
# getAngle(pos : Position, dir : int, gpos : Position) : int
# getAltitudeDiff(z : int, gz : int) : int
<<data>>
Characteristics
~ leftMap : map<int,int>
~ rightMap : map<int,int>
~ upperMap : map<int,int>
~ distMap : map<int,int>
<<data>>
MotorPower
~ left : int
~ right : int
~ upper : int
<<data>>
Position
~ x : int
~ y : int
~ z : int
<<data>>
Goal
~ pos : Position
~ period : int
<<data>>
USSensor
- id : int {id}
~ pos : Position
- plan
- system
- blimp
- writer
- reader
- altSensor
- dirSensor
- contoroller
- lock
- posSensor
- timer
- logger
- motor
- sender
- mode

最終版クラス図(1/1)