衛星開発ゼミ

WEEK 7

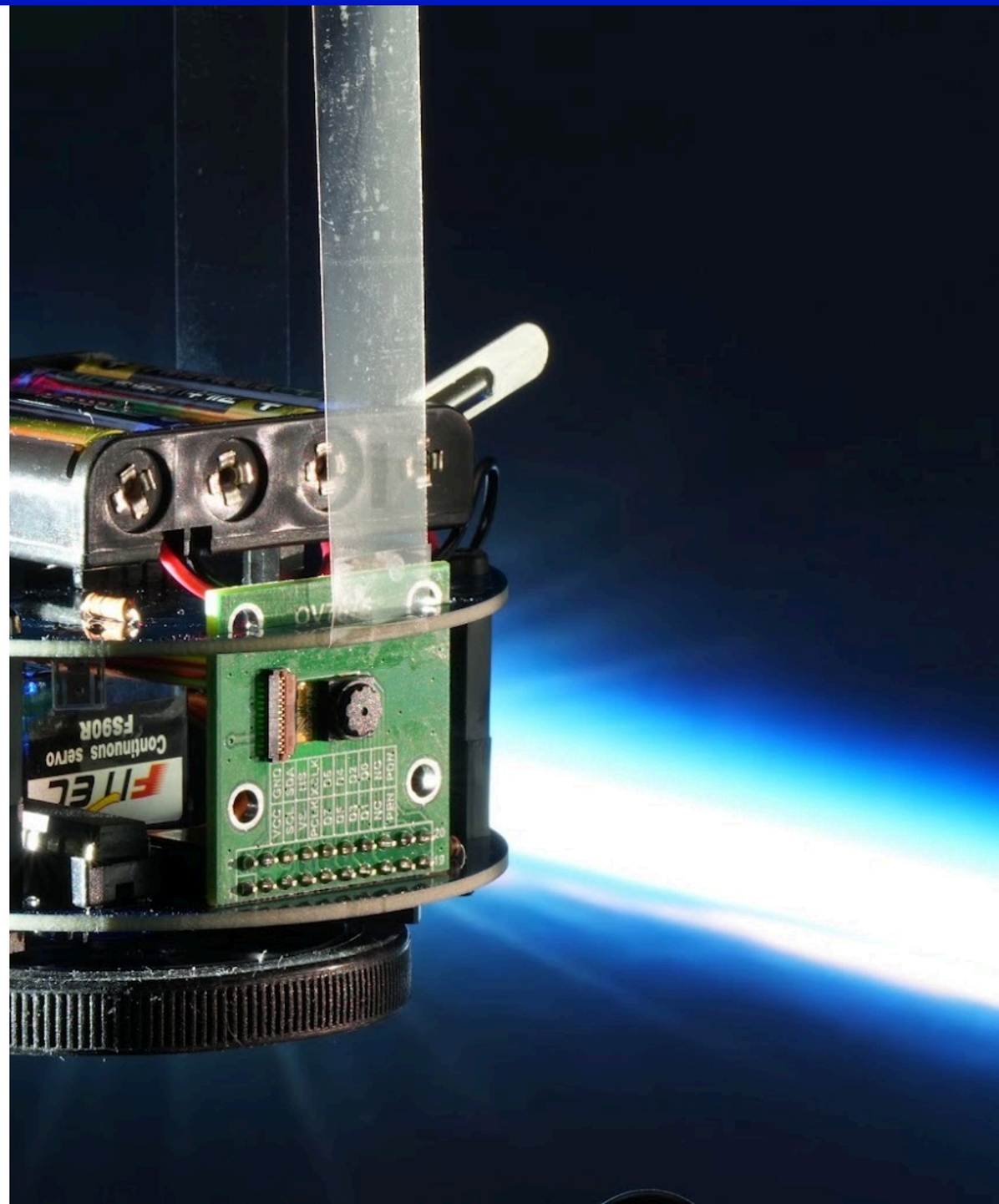
ソフトウェア エンジニアリング

良いソフトウェア設計とは、
変更を安全かつ高速に回せる構造を作ることである

軸① 設計：責務を分け、依存先を実装からインターフェースへ移す

軸② 検証：PC 上で動かし、Unit Test・TDD・SILS で確かめる

軸③ 統合：小さな変更に、自動チェックとレビューをフィードバック

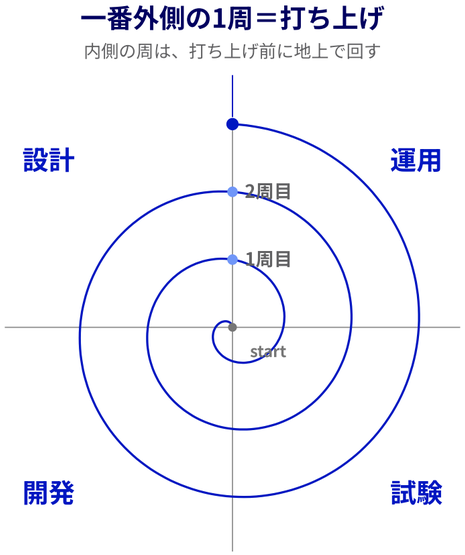


0.1 前回の講義から

完成形を一度で設計せず、検証できる小さな単位から始める

前回の宿題（再掲）　**模擬衛星の初回打ち上げに必要な機能を考える**

- **完成形の設計は、一度では決まらない。**
  要求から具体的な設計を一足飛びには導けない。作ってみて初めて分かることがある。
- **まず、検証できる小さな範囲を切り出す。**
  到達点まで全部を作るのではなく、今日作って結果を観測できる単位まで小さくする。
- **小さく作り、小さく確かめる。**
  設計 → 実装 → 試験 → 観測を一周させる。
- **分かったことを、次の設計へ戻す。**
  一周で得た知見を使って、次の一周を決める。



今日学ぶのは、ソフトウェア開発でこのサイクルを小さく・速く・安全に回すための技術。

1

なぜソフトウェアエンジニアリングが必要か

ソフトウェア開発では、変更と学習を繰り返す。
衛星では打ち上げ前に、そのサイクルをできるだけ高速に回しておく必要がある。

1.1

設計は一度では決まらない

設計そのものが動き、変更コストも物理に
縛られないので、改善し続けられる

1.2

実機では何度も試せない

確かめたいことのすべてを、実機で
繰り返し試すことはできない

1.3–1.4

何度でも試せる場所を作る

実機より速く、何度でも試せる場所を
ソフトウェアの構造として用意する

設計は一度では決まらない。そしてソフトウェアは、それを前提に作れる

システム開発一般

実装して初めて分かることがある

責務の切り方、依存の向き、必要な情報などは、書いて動かして初めて見える

ソフトウェア固有

設計そのものが動く

コードは設計書と製品が同一物。製造工程がないので、設計をそのまま実行して確かめられる

ソフトウェア固有

変更コストが物理に縛られない

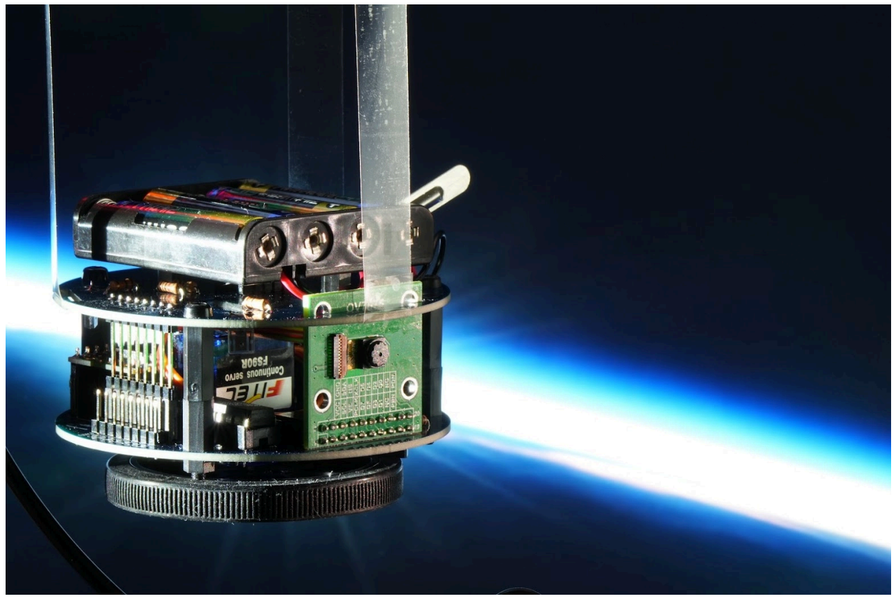
複製・巻き戻し・やり直しが安い。同じ検証を何度でも繰り返せる

↓ だから

重要なのは、最初から完璧に設計することではなく、設計を速く安全に改善できること

衛星では、確かめたいことのすべてを実機で何度も試すことはできない

- **開発中は自由回転させられない。**
  Week 6 の運用ルール。ベアリングで自由回転させるのは打ち上げの中だけ
- **軌道上と同じ条件を地上で完全には再現できない。**
  実衛星も同じで、打ち上げ前に作り込むしかない
- **実物での試行錯誤には制約がある。**
  試せる回数・条件・機体への負荷が限られる



具体例：姿勢制御。開発中は機体を固定し、自由回転はさせない

設計を何度も改善するにはフィードバックが必要。しかし、そのフィードバックを毎回実機から得ることはできない。

広く・現実に近いことを確かめるほど、1周は遅くなり試せる回数も限られる。だから、手前の段で確かめられることは手前の段で確かめる

確かめる場所	1周にかかる時間	要るもの
静的解析	秒未満	コードだけ
Unit Test	秒	PC だけ
SILS	秒〜分	PC ＋ 機体モデル
HILS	分〜時間	実物の一部 （マイコン・ホイール）
実機統合試験	時間〜日	機体一式と人の立ち会い
打ち上げ	一度きり・やり直せない	本番の機会そのもの

さらに、人間が毎回手で実行・目視していては高速には回らない。手前の段の検証は、繰り返し実行できる形にする。

実機依存と人手依存の両方を減らす。青い3段を自分で作るのが今日の目標

1.4 今日学ぶことの全体像

どうすれば、変更の結果をもっと早く確かめられるか

実機では何度も試せない　→　何度でも試せる検証環境を地上に作る必要がある　→　そのために、設計・テスト・SILS・CI・構成管理を学ぶ

設計

2章　変更に強い構造

責務・凝集度・結合度・
情報隠蔽・インターフェース

→

検証

3章
テスト
可能性

Unit Test

4章
小さく
直す

TDD・
Refactoring

5章
全体を
試す

MBD・SILS

→

統合と再現

6–7章
小さく統合

GitHub Flow・
Review・CI

8章
再現できる

構成管理・
Release

AI と人間

AI は設計・実装・検証環境まで作ってよい（設計案・実装・テスト・SILS・CI の設定まで）。
人間は Evidence の十分性・独立性と Risk を見て Accept / Revise / Reject を決め、最終的な mission acceptance の責任を持つ。

すべて、変更→実行→観測→修正のループを短くし、変更を安全にするために必要になる

実機のフィードバックは遅く、試せる回数も限られる。だから、地上に速く何度でも確かめられる場所を作る。

①

設計は一度では決まらない

実装して初めて分かることがある。
ソフトウェアは設計そのものが
動き、変更コストも物理に縛られない

②

実機では何度も試せない

試せる回数と条件に制約があり、
軌道上と同条件の地上試験もできない

③

確かめる場所を手前の段へ移す

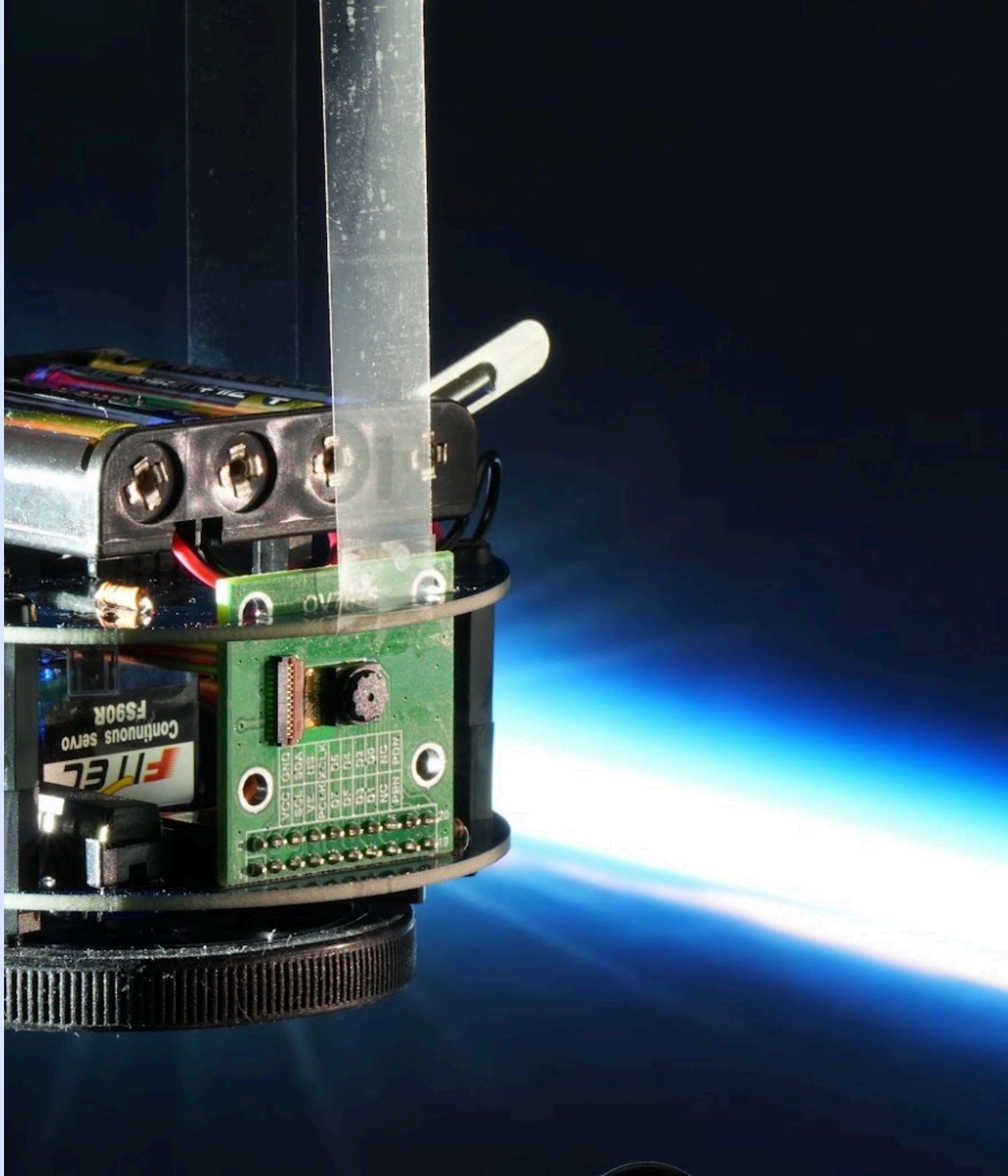
静的解析は秒未満、Unit Test は秒、
SILS は秒〜分、実機統合試験は時間〜日。手前で確かめられることは手前で
確かめ、繰り返し実行できる形にする

次章へ　手前の段で確かめるには、まずコードの構造がそれを許す形になっていなければならない

2

変更に
強いソフトウェアを
設計する

一つの変更がどこまで波及するかは、設計で
決まる。責務を分け、変わりやすい判断を隠し、
依存先を実装からインターフェースへ移す。



メインループでベタ書きすると、役割の違うコードが同じ場所に混ざる

```
// 直書きの例：判断ロジックとハード操作は違う役割
while (true) {
    double sun_angle = read_sun_angle();    // ← ここで実機の ADC に直接触っている
    if (sun_angle > kThreshold) {
        set_wheel_command_us(1600);         // ← ここで実機のサーボに直接触っている
    }
}
```

- read_sun_angle() が実機の ADC を直に叩くため、ADC が無い PC では動かない
- 「太陽方向で回転を決める」という判断ロジックだけを取り出せない

依存が多く深いほど、変更は広く波及する

依存

A が B に依存する＝B の変更が A に波及しうる

必要な依存はある。減らしたいのは、相手の内部まで知っているという不要な依存

↓

結合度

依存先が多く、相手の内部を深く知るほど高結合

直書きコードの場合（高結合）

判断が、センサの読み出し手順や生の値の形式まで知っている。
センサ側の小さな変更が判断に波及し、判断だけを取り出して試せない

↓

この章の方針

部品どうしは不要な依存を減らし（低結合）、部品の内側は同じ責務でまとめる（高凝集）

独立して変更されそうな設計判断を、モジュールの内側に隠す

- **独立して変更されそうな判断を隠す。**
  ADC の分解能、配線、電圧から角度への
  変換式。他の部分と関係なく変わりうる判断
- **外に見せるのは約束だけ。**
  太陽方向を返す、という一行だけを公開する
- **隠せた範囲が、変更で読み直さなくてよい範囲になる。**
  漏れていれば、呼び出し側すべてが変更の対象になる

公開する約束

```
read_sun_angle() : double
```

↓

モジュールの内側に隠す判断

- ADC のチャンネル割り当て
- 分解能と基準電圧
- 電圧→角度の変換式
- 平滑化の方法
- 校正値（calibration）

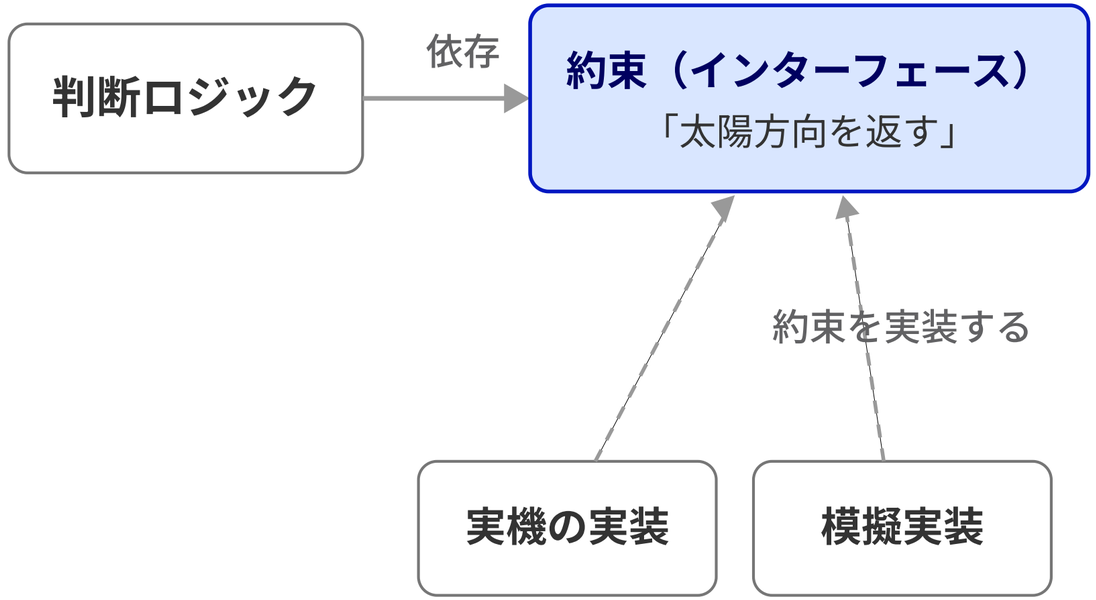
ここが変わっても、判断ロジック側は書き換えなくてよい

依存の相手を、具体的な「実装」から「インターフェース（守るべき取り決め）」に変えると、中身を差し替えられる

いま：実装に直接依存する

ADC が無い場所では、判断ごと動かない

これから：インターフェースにだけ依存する

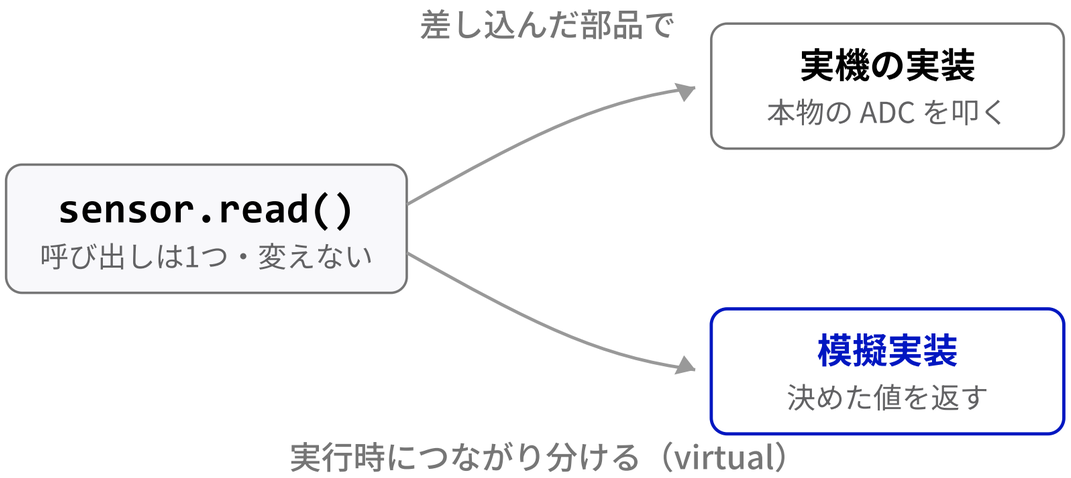


判断が見るのは約束だけ。中身は差し替え自由

この間接化は言語を選ばない（C では関数ポインタでも作れる）。今回は **C++ のクラス**を道具に使う

INFO　索引：Interface を C++ で実現する手段の一つ（必要になったときに引く）

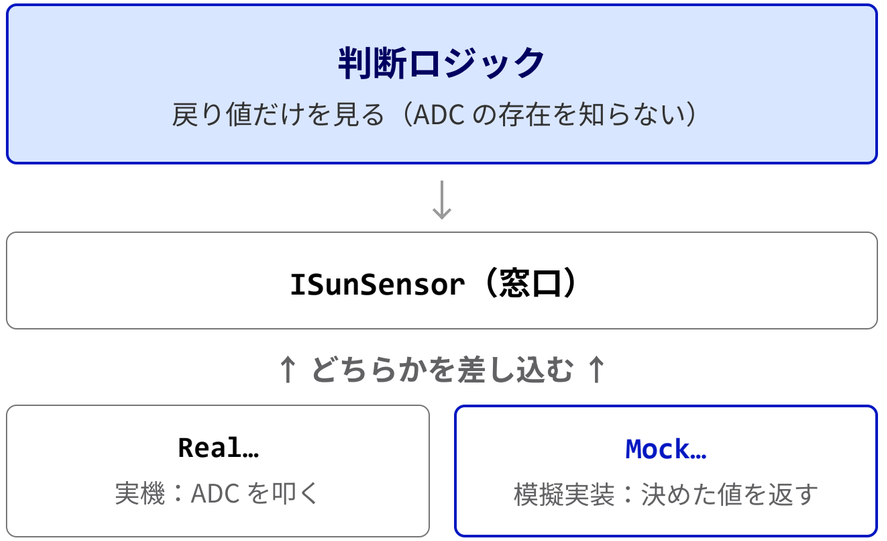
C++ では、クラス＋virtual が差し替えを実現する方法の一つ



- **クラス。**
  データと操作を1つの部品にまとめる。窓口も実装もクラスで表す
- **仮想関数（virtual）。**
  同じ呼び出しで、実機と模擬実装の
  どちらが動くかを実行時に切り替える
- **設計思想と実現手段は別。**
  Interface は設計の考え方、virtual はその実現手段の一つ（テンプレートや関数ポインタなど他の手段もある）。文法の細部は必要になったときに引けばよい

判断を「太陽方向を返す何か」に依存させ、実機と模擬実装（Test Double）を差し替える

```
// 判断は「太陽方向を返す窓口」にだけ依存する
class ISunSensor {          // 窓口（インターフェース）
public:
    // 「太陽方向を返す」とだけ約束する
    virtual double read_sun_angle() = 0;
};
// 実機：ADC を叩く
class RealSunSensor : public ISunSensor {
    double read_sun_angle() override;
};
// 模擬実装（Test Double）：決めた値を返す
class MockSunSensor : public ISunSensor {
    double read_sun_angle() override;
};
```



PC のテストでは模擬実装に決めた角度を入れて判断を確かめる

良い分離は、変更の影響範囲と試験の範囲を小さくする。

①

部品の内側は高凝集、部品どうしは低結合

責務で分け、変わりやすい判断は情報隠蔽でモジュールの内側に置いた

②

依存の相手を実装からインターフェースへ変えた

インターフェースに依存させたので、
実機センサと模擬センサを差し替えられる

次章へ　PC 上でロジックを動かせるようになった → 次は入力と出力を与えて自動で確かめる

3

テストできることは設計品質である

分離した結果、判断ロジックはPC上で動く。入力と出力を
与えられるなら、その振る舞いは自動で確認できる。

3.1

入力と出力を与える

模擬実装で入力を決め、
出力をコードで確かめる

3.2

Testability は設計の性質

テストしにくさは、書き方ではなく設計の問題

3.3–3.4

何をどこまで試すか

同値分割・境界値分析と、Coverage の読み方

入力を決められて出力を読めるなら、その振る舞いは自動で確認できる

```
// 明るい側ではホイールを回す、という期待を書く
TEST(SunPointing, TurnsWheelWhenBright) {
    MockSunSensor sun(30.0);   // 入力を決める
    SunPointingLogic logic(&sun);
    auto cmd = logic.step();     // 出力を受け取る
    EXPECT_EQ(cmd.wheel_us, 1600);
}
```

- **入力は模擬センサで決める。**

  MockSunSensor に角度を持たせるだけで、実機の ADC は不要

- **出力は戻り値で読む。**

  ホイールへの指示が値として返るので、期待と機械的に比べられる

- **Test Double（模擬実装）。**

  決めた値を返すもの、簡易な実装を持つものなどの総称。
  配布テンプレートでは MockSunSensor という名前で用意している

- **確認が資産になる。**

  一度書けば、変更のたびに同じ確認を同じ手順で繰り返せる

Testability はテスト担当者の都合ではなく、ソフトウェア設計そのものの性質である

Testability の中心は3つ

入力を決められる
初期状態と入力を思いどおりに与えられる（Controllability）

結果を見られる
必要な出力と状態を外から観測できる（Observability）

同じ条件を再現できる
同じ条件なら同じ結果になる（Repeatability）

PC 上で動かせることは、この3つを手軽に実現する有力な手段。Testability ＝ PC で動くこと ではない

テストしにくいコード

- ✕ ハードウェア入出力と判断が同じ関数に同居している
- ✕ 時刻・乱数・通信の応答で結果が変わる
- ✕ 結果が内部状態にしか現れず、外から読めない
- ✕ 確かめるには実機を用意して人が見るしかない

テストしやすいコード

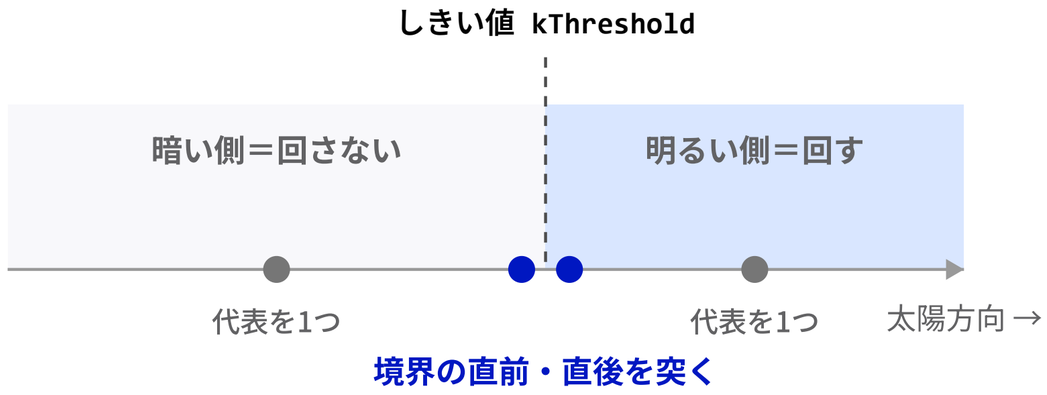
- ○ ハードウェア入出力などの副作用を、境界の外へ追い出している
- ○ 同じ初期状態と入力列なら、同じ結果を再現できる
- ○ 入力は引数、結果は戻り値として外から読める
- ○ 模擬センサを差し込めば PC 上で完結する

テストが書きにくいときは、テストの書き方ではなく設計を疑う

入力は、同値分割で代表を選び、境界値で誤りの出やすい点を突く

- **同値分割。**
  同じ振る舞いになる入力のまとまりから、代表を1つずつ選ぶ
- **境界値分析。**
  しきい値のちょうど・直前・直後を
  突く。比較の向きの誤りはここで出る
- **異常系も1つは書く。**
  「30°なら1600を返す」だけでなく、センサ更新が止まったら
  （timeout・stale data）どう振る舞うか。範囲外値・
  NaN・アクチュエータ飽和も同じ形で期待を書ける

テスト入力の選び方（例：太陽方向のしきい値）



比較の向き（> と <）の取り違えは、境界のテストが捕まえる

Coverage が高いだけでは正しさは言えない。見えるのは「どこまで試したか」

言えること

試した範囲が見える

実行された命令や分岐の割合が出る。命令網羅（C0）・分岐網羅（C1）という段階がある

言えないこと

正しさは出てこない

試していない入力には何も言えない。
期待値そのものが間違っていれば緑になる

使い方

薄い場所を探す道具

数値を上げる目標ではなく、確認が
届いていない箇所を見つけるために読む

- **テストが積み上げるもの。**
  定義した期待動作を継続的に確認できること、そして変更で壊れたこと（Regression）を早く検出できること
- **テストが積み上げないもの。**
  そのテストが対象にしていない性質、そしてシステム要求そのものの妥当性（→ Systems Engineering の回）

INFO　発展：テストの感度を確かめる（Mutation Testing）

意図的に壊して赤になるかを見ると、何も見ていないテストを見つけられる

STEP 1

テストが緑で通っている

ただし、これだけでは「甘くて何も見ていない」場合と区別できない

→

STEP 2

対象をわざと1か所壊す

Week 3 の「わざと壊して壊れ方を観測する」のテスト版

→

STEP 3

テストが赤くなる

その壊し方には反応する、という確認

言えるのは「その壊し方には反応する」まで。それでも、何も見ていないテストは見つかる

テストできること自体が、ソフトウェア設計の品質である。

①

確認する作業をコードにした

入力を決められて出力を読めるなら、その
振る舞いは自動で確かめられる（Unit Test）

②

テストしやすさは設計の性質

入力を決められる（Controllability）
結果を見られる（Observability）
同じ条件を再現できる（Repeatability）

③

Coverage は物差しであって保証ではない

分かるのはどこまで試したか。積み
上がるのは継続的に確認できるという根拠

次章へ　数秒で確認が返るなら、設計を小刻みに直しても壊さずに済む

4

小さく設計し、小さく直す

テストが数秒で返るなら、設計・実装・検証を非常に
小さな単位で往復できる。最初から完璧な設計は要らない。

4.1

赤→緑→リファクタ

期待を先に置き、1周を数分の単位で回す

4.2

なぜ小さく往復するのか

外した場所を直前の一歩に閉じ込める

4.3

Refactoring

振る舞いを変えずに内部設計を改善する

次の一歩を、検証できる大きさまで小さくする

大きく作ってから確かめる進め方

まとめて実装する
まとめてテストする
問題が出たら広く切り分ける

・一度に足した分がまとまって効くので、外した場所の候補が広がる

・自動テストの有無ではなく、1回の
フィードバックが受け持つ範囲の大きさが違う

TDD：小さく往復する

赤

次の期待を1つ書く。

まだ満たされていないため、意図どおり失敗することを確認する

↓

緑

最小限で通す。

↓

リファクタ

確認してから Refactor し、次へ。

次のテストへ戻り、1周を繰り返す。確認はコードとして残り続ける

設計と実装と検証を、非常に小さな単位で往復する開発の進め方

大きく作ってから確かめる

外した場所の切り分けが広くなる

変更がまとまっていると、どこが原因かを
切り分ける範囲が広がり、手戻りも広がりやすい

小さく往復する

原因の探索範囲を狭めやすい

赤になった時点で、原因の探索範囲を直前の
変更周辺に狭めやすい。手戻りも局所化しやすい

- **1周は数分、振る舞い1つ。**
  大きく作ってから確かめるのではなく、確かめられる大きさに設計を刻む
- **先に書く理由は2つ。**
  期待をコードで表してから実装に入り、その期待がまだ満たされていないことを Red で確認する
- **言い過ぎない。**
  テストは要求仕様そのものではない。テスト対象となる振る舞いの一部を、実行できる例として表したもの

外部から見える振る舞いを変えずに、内部設計を改善する

- **意図は内部構造の改善。**

  外部から見える振る舞いを変えずに内部を改善する。
  緑が崩れたら、それは Refactoring ではなく変更

- **Green は Evidence の一つ。**

  Green なら、少なくともそのテストが観測している
  振る舞いについて Regression は見つかっていない。
  それを数秒で得ながら構造を変えられる

- **やることは小さい。**

  名前を直す、重複を抜く、責務を分ける、窓口を切り出す

固定される

外から見える振る舞い

テストが表している入出力。ここは動かさない

自由に変えられる

内部の構造

責務の切り方、関数の分け方、名前、データの持ち方

最初から完璧なアーキテクチャを設計する必要は
ない。**小さく作り、確かめ、学び、設計を改善する。**

小さく確かめられるから、設計は後から安全に改善できる。

①

一歩を、確かめられる大きさまで小さくする

期待を1つ書いて意図どおり赤になるのを見て、通す
最小限で緑にし、緑のまま整えてから次へ進む（TDD）。
赤が出た時点で、原因は直前の一歩の周辺に絞られる

②

Refactoring は振る舞いを変えない改善

Green は、テストが観測している範囲で Regression が
見つかっていないという Evidence。変えるたびにテストを
走らせ、その Evidence を確かめながら構造を良くする

次章へ Unit Test で確かめたのは部品単体の入出力だけ。部品を全部つないで動かしたときの振る舞いはまだ分からない

5

実機なしでシステム全体を試す

Unit Test で確かめたのは部品単体まで。次は、部品を全部つないで動かしたときの振る舞いを確かめる。実機の代わりに、実行できるモデルの中で制御ループを閉じる。

5.1

検証の階層

Unit / Integration → SILS
→ Target → HILS → 実機

5.2–5.4

実行できるモデルと SILS

モデルを実行し、その中に実際の
フライトロジックを接続する。用語より、
実機を使わず何度も回せることが本質

5.5–5.6

モデルと実機のずれ

パラメータは実機で測り、
残ったずれはテレメトリで直す

段を上げるほど範囲は広がる。段ごとに得られる Evidence の種類が違う

段	その段で得られる Evidence	その段では得られないもの
静的解析	動かさずに分かる危ない書き方	実際の振る舞い
Unit Test Host（PC）上	部品単体の入出力が期待どおりか 局所化と再現性。数秒で返る	部品どうしをつないだときの振る舞い
Integration Test Host（PC）上	複数部品の接続が成立しているか 例：SunSensor 実装と判断ロジックをつなぎ、値が正しく渡るか	機体が実際にどう動くか
SILS	制御ループ全体の振る舞い（姿勢が収まるか）	モデルと実機のずれ・target 固有の性質
Target 試験 On-target Test	実 CPU・compiler / ABI・RTOS 上での timing / deadline interrupt・scheduling・CPU load・memory / stack・queue・bus 帯域。PC 上では再現しにくい	実ハードとの interface・実環境
HILS	実ハードとの interface、タイミングや応答	軌道上・自由回転の環境
実機統合試験	実ハード全体の統合挙動	軌道環境・自由回転などは残る
打ち上げ・運用	実際の環境での振る舞い	やり直しが極めて困難

手前の段で確かめられる性質は、その中で最も信頼できる段で早く潰す。
ただし、その段でしか得られない Evidence は省略しない

対象の振る舞いをモデルにすると、設計を実行して比べられる

- **モデルとは。**
  機体の運動や機器の応答を、数式とコードで表したもの
- **実行できることの意味。**
  設計案を回して結果を比べられる。紙の上の議論が観測に変わる
- **押さえるのはここ。**
  実行できるモデルを使えば、変更→実行→観測→修正を実機を使わずに何度も回せる。覚えるのは用語ではなくこの効果

この講義での MBD

実行可能なモデルを設計・検証に使う考え方として紹介する

MBD の全体系はこの講義では扱わない

SILS（構成の一つ）

モデル環境に実際のフライトロジックを接続して検証する

PC 上の仮想的なセンサ・アクチュエータ・機体モデルへつなぐ。
SILS は MBD を採用していなくても構築できる

機体の物理を模す Plant と、判断を担う Controller を分け、PC 上でループさせる

- **Controller。**
  センサ値を受けて、モータへの指示を決める（実機に載る頭脳）
- **Plant。**
  モータ指示を受けて、機体の回転と次の
  センサ値を計算する（実機の代わり）
- **2章の分離が効く。**
  同じ Controller を、実機にも SILS にも差し替えるだけで載せられる

SILS が作れるのは、フライトロジックがハードウェアから分離されているからである

低結合・高凝集

↓

インターフェースで依存を切る

↓

ハードウェアを差し替えられる

↓

ロジックが PC 上で実行できる

↓

SILS が成立する

- **差し込む先は同じ窓口。**

  Unit Test の模擬センサと、SILS の Plant は、同じインターフェースの向こうに入る

- **Controller は書き換えない。**

  同じフライトロジック、可能なら同じ source を SILS でも使うことで、実機へ近いロジックを地上で検証できる

- **逆も真。**

  ハードウェア依存が分離されているほど、Unit Test や SILS を小さく・局所的に構築しやすい。設計が検証環境の前提になっている

測れる値は実測し、残った差を観測してモデルを更新する

実測値と仮定値を区別する

実測しないと決まらない値

機体とホイールの慣性　指令→トルクの換算　軸受の摩擦

正しい数は運営からは渡らない。各チームの機体の中にしかない

仮定として置く値

摩擦モデルの形、無視する外乱、時間の
刻み方。置いた仮定はそのまま残差になる

残った差はテレメトリに現れる

SILS の予測：こう動くはず

↓

打ち上げのテレメトリ：実際はこう動いた

↓

差を見てモデルを直す

ずれは失敗ではなく前提。Week 6 の観測設計は、モデルを直す材料でもある

分離してあるから、同じフライトロジックを PC 上のモデルの中でも回せる。

①

確かめたい性質に応じて、最も手前で信頼できる段を選ぶ

Unit / Integration → SILS
→ Target → HILS → 実機。
段ごとに得られる Evidence の種類が違い、その段でしか得られない Evidence は省略しない

②

Plant と Controller でループを閉じる

同じフライトロジック（可能なら同じ source）を Host（PC）上で回して姿勢制御を作り込める。compiler・ABI・RTOS・timing の差は Target 試験で確かめる

③

モデルは現実そのものではない

実測しないと決まらない数（慣性・換算・摩擦）は測る。残ったずれは打ち上げのテレメトリで更新する

次章へ 1人で変更→検証を速く回せるようになった。次は、その小さな変更をチームで壊さず合流させる

6

変更を小さく統合する

1人で速く回せるようになった変更を、チームで壊さず合流させる。
ここでの「小さく」は設計の話ではなく、一度に持ち込む変更量の話。

6.1

GitHub Flow とは

main を動く状態に保ち、短命な
枝と Pull Request で変更を戻す型

6.2

小さく保つ

短命なブランチと小さな commit。
長く育てた枝は統合が難しくなる

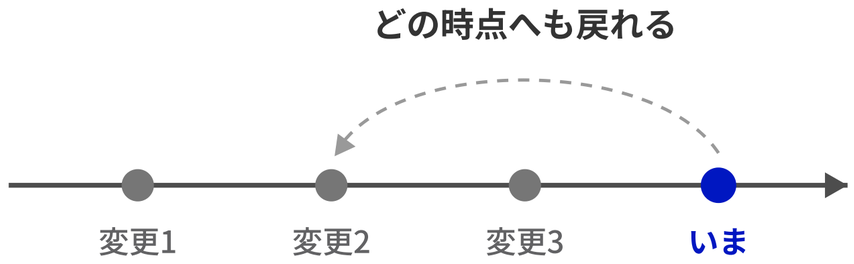
6.3

差分にフィードバック

Pull Request で単位を示し、
Code Review で差分を読む

バージョン管理は変更の履歴を残し、どの時点へも戻れるようにする仕組み

バージョン管理	**考え方。**変更を履歴として記録し、いつでも過去の時点へ戻れるようにする
Git	**道具。**履歴とブランチを扱う。Week 0 で「作業を戻せる場所を作る道具」と位置づけたもの
GitHub	**置き場。**Git の履歴をチームで共有する場所。Pull Request などの共同作業の仕組みが載る



変更のひとつひとつが、戻り先になる記録として残る

main を常に動く状態に保ち、変更は短命な枝で作って Pull Request で戻す

1
branch を切る
main から枝分かれ。何をする枝か分かる名前を付ける

→

2
commit する
変更を小さく刻んで、意図ごとに履歴へ残す

→

3
Pull Request を開く
この差分を main に入れたい、という単位を示す

→

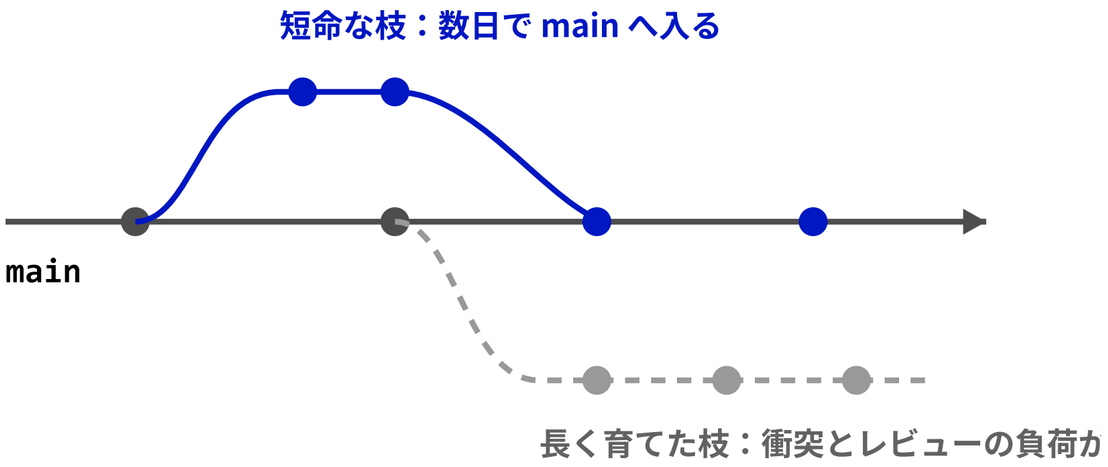
4
Review と自動チェック
人が差分を読み、機械がビルドとテストを走らせる

→

5
merge して枝を消す
main へ入れたら枝は捨てる。次の変更はまた main から

- **枝は main からだけ生える。**
  長期の develop や release ブランチを並走させない。履歴の中心は常に一本の main である
- **main は壊さない、が唯一の約束。**
  main はいつ取り出しても動く状態に保つ。だから確認は merge の前、Pull Request の上で済ませる

長く育てた巨大なブランチは統合が難しくなる。変更は小さく刻んで持ち込む



- **ブランチは作業単位の器。**

  main（最新の統合基準線）から枝分かれさせた独立の作業場。ここでの「小さく」は2章の影響範囲ではなく、一度に持ち込む変更量の話

- **短命に保つ。**

  長く生きた枝は他の変更と衝突し、差分が大きすぎて読めない。統合そのものが難しくなる

- **commit も小さく。**

  1コミット1つの意図。何をしたかが履歴で読めて、戻す単位にもなる

PR は単なる関所ではない。変更単位を明示し、差分へフィードバックを集める場所である

- **PR は提案である。**

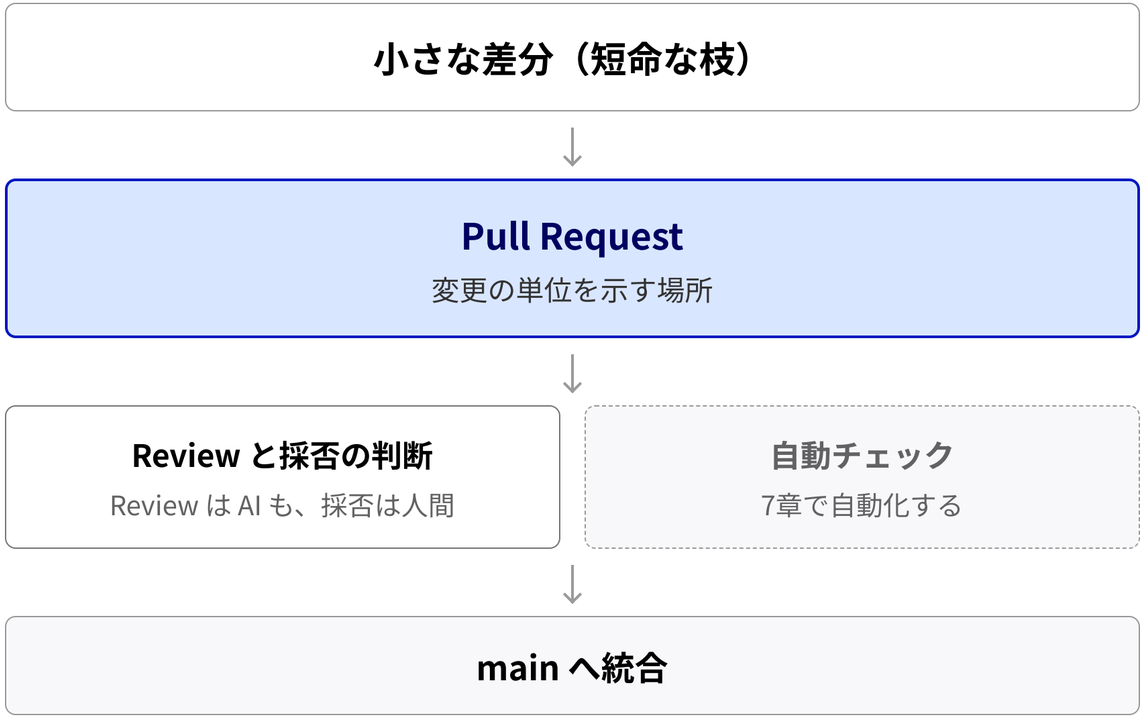
  この差分を main に入れたい、という
  単位の明示。何を変えたかが1画面に収まる

- **Review は別視点からの指摘である。**

  変更に対して問題・懸念・改善案を出す活動。AI が
  実行してもよく、人間は挙がった論点を見て採否を決める

- **小さい PR ほどレビューが効く。**

  大きすぎる差分は論点が埋もれ、形式的な承認しか生まない



設計では影響範囲を小さくし、統合では一度に持ち込む変更量を小さくする。

①

短命なブランチと小さな commit

長く育てた巨大な枝は他の変更と衝突し、レビューも統合も難しくなる

②

小さな差分を頻繁に main へ入れる

Pull Request で変更の単位を示し、Review で別視点のフィードバックを受けてから統合する。差分が小さいほどレビューが効く

次章へ テストがあっても、人間が走らせるならそこが手作業として残る

7

フィードバックを自動化する

テストがあっても人間が走らせるなら、そこが手作業として
残る。変更した直後に、機械が品質のフィードバックを返す。

7.1–7.2

CI とは何か

共有の場所が、変更のたびに同じ手順で
build・静的解析・テストを走らせる

7.3–7.4

自動チェックと判断、二重のループ

根拠を積むのは機械、採否を決めるのは
人間。自動ループは数分で返る

7.5

1章の問いへの答え

変更の結果を、もっと早く
確かめられるようにする

人が思い出して自分の PC で確かめる代わりに、共有の場所が変更のたびに同じ手順で確かめる

名前の意味

Integration は、自分の変更をチーム共有の main へ合流させること。**Continuous** は、それを小さく頻繁に繰り返すこと

1回の CI で起きること

1 **自分の変更を push する（または PR を出す）**

いつもの作業。ここから先に人の操作は要らない

2 **共有サーバが変更を検知し、自動で確かめる**

まっさらな環境を用意し、設定ファイルに書いた手順を上から実行する

3 **緑・赤とログが、その変更の画面に返る**

どこで失敗したかまで残る。赤ならその場で直してもう一周する

- **走るのは自分の PC ではない。**

  GitHub などの共有サーバ上の自動実行（GitHub Actions など）。誰の変更でも、同じ場所で同じように確かめられる

- **毎回まっさらな環境で実行する。**

  手元にだけ入っていたライブラリや設定に頼れないので、「自分の PC では動く」が起きにくくなる

- **手順は設定ファイルに書いて履歴に残す。**

  どのテストを走らせるかを人が覚えておく必要がなくなる。手順そのものもレビューできる

- **目的は合流のたびの確認。**

  テストを自動で走らせること自体が目的ではなく、頻繁な合流のたびに根拠を受け取ることが目的

変更をきっかけに走らせるものを決め、その結果を差分の上に返す

変更を小さくする → 短命な枝 / small batch → 統合基準線へ頻繁に統合 → そのたびに自動 Feedback

Push / Pull Request をきっかけに走るもの

build	そもそも通るか
compiler warnings	警告は消してから進む
formatter	書式を自動で整える
lint / static analysis	規約違反や怪しい記述を、動かさずに指摘する
unit test	部品単体の期待動作
integration test	部品どうしの接続
SILS scenario	必要なら、制御ループごと回す

- **引き金は変更そのもの。**
  人間がコマンドを打たなくても、push や PR が機械を走らせる
- **結果は差分の上に返る。**
  required checks を満たしたかが出る。合流はレビューと判断のあと
- **覚えておく必要がなくなる。**
  どのテストを走らせるべきかを人間が記憶する仕事が消える
- **統合のたびに返る。**
  小さな差分を main（最新の統合基準線）へ頻繁に統合し、その統合ごとに Feedback を受ける

CI は「自動テストサービス」ではなく、Frequent Integration ＋ Automated Feedback。変更を共有される統合基準線へ頻繁に入れ、そのたびに build / test / analysis が根拠を返す

自動チェックは根拠を返し、判断はその根拠を採用するかを決める

CIで実行する自動チェック（静的解析・Unit Test・Integration Test・SILSなど）

- 決められた基準に対して、高速に・再現可能に結果を出す
- 変更直後に根拠を返す。何度でも同じ手順で繰り返せる
- 基準の外側には何も言わない
- 基準そのものが妥当かは判定できない

Review と Decision（差分レビューと採否の判断）

- 根拠と設計案を見て、基準・リスク・採否を決める
- 抜けた観点や設計の筋に気づける。
  Review は AI も支援・実行してよい
- 採否の判断は自動チェックの外側に
  あり、小さい差分でなければ働かない
- AI はレビュー案や改善案を作ってよい。採否は人間が持つ

分けるのは AI か人間かではなく、根拠を積む自動チェックと、それを採用するかの判断である

自動チェックは数分で戻り、統合は判断を経て進む

内側：自動ループ（数分）

変更 → Build → 静的解析 → Unit Test → SILS → 根拠が返る

← 赤ならその場で直して、もう一周する

外側：統合のループ（判断を含む）

Pull Request → 自動チェックの結果 → Code Review → 人間の判断 → Merge

Code Review は数分では返らない。だから自動ループを内側に置き、判断は差分の単位でまとめて受ける

1章の問いへの答え：手前の段に自動ループを作れば、変更の結果は数分で返る

変更した直後に、数分で根拠が返る状態を作る。

①

CI は自動テストサービスではない

Frequent Integration ＋ Automated Feedback。小さな差分を統合基準線へ頻繁に入れ、そのたびに build・静的解析・Unit Test・SILS が根拠を返す

②

二重のループで受ける

内側の自動ループは数分で返る。外側の統合ループは Review と人間の判断を経て進む

③

返るのは保証ではなく継続的な確認

自動チェックが根拠を積み、それを採用するかの意思決定は人間が持つ

次章へ CI の Green は「この構成で確かめた」という Evidence。では、実際に飛ばした構成は何だったのか

8

飛ばしたソフトウェアを再現できるようにする

テストが緑でも、軌道上で動いていたものが何だったかを言えなければ、原因は追えない。

8.1

なぜ構成管理が必要か

同じものを手元で動かせなければ、
異常の切り分けが始まらない

8.2

最低限やること

Release と Tag、依存の
固定、build artifact の保存

帰結

戻れることは副産物

再現できる状態が
保たれているから、外れても戻れる

同じものを手元で動かせなければ、原因の切り分けは始まらない

前提 **軌道上で異常が出る**

手元にあるのはテレメトリだけで、実機は取り戻せない

↓

必要 **原因を絞るには、同じものを手元で動かす**

飛んだものと同じコード・同じ設定・同じ焼き方で再現して、初めて比べられる

↓

帰結 **「何が載っていたか」が言えないと、そこで止まる**

どこが違うのか分からないので、切り分けも、直したことの確認もできない

だから、飛ばしたものを後から再現できる状態を、打ち上げ前から保っておく

8.2 Release・Tag・依存固定・artifact 保存

時点を指せて、同じものを作り直せるなら、原因を追えて戻れる

STEP 1

Release / Tag

この時点が飛んだ、と後から
指せる名前を付ける

STEP 2

Dependency 固定

ライブラリとツールの版を
書き残し、同じ環境を
作り直せるようにする

STEP 3

Configuration の版管理

しきい値や校正値も、コードと
同じように履歴へ載せる

STEP 4

Build artifact の保存

実際に焼いたバイナリ
そのものを残す

「外れたら戻せる」は、再現できる状態が保たれていることの副産物である

「何を飛ばしたか」を再現できて、初めて原因を追える。

①

構成管理の対象は commit だけではない

toolchain、依存のバージョン、設定と校正値、焼いたバイナリまで含む

②

その時点を指して同じものを作り直せる

外れたら戻せるというのは、この状態が保たれていることの副産物

次章へ 変更を安全・高速に回すという一つの思想から、設計・検証・統合がどう出てきたかを一枚で見る

9

まとめ：高速なフィードバックループを作る

今日の技法はバラバラのベストプラクティスではない。変更を
小さく、安全に、高速に回すという一つの思想から出てくる。

9.1

一つの思想から三つの仕組みへ

設計・検証・統合が、並行して
フィードバックループを支える

9.2

結論

完璧に作る技術ではなく、
改善し続けるための技術

9.3–9.4

次回予告と宿題

Week 6 の宿題を、今日の道具で考え直す

「変更を安全・高速に回す」を、設計・検証・統合の三つで支える

↓

設計｜2–4章	検証｜3・5章	統合｜6・7章
責務分離・情報隠蔽	Static Analysis	Small batch
Interface	Unit / Integration（Host）	Short-lived branch
Testability	SILS / Target	PR / Review
Fast Unit Test → Refactoring	HILS / 実機	Frequent Integration
	Evidence を早く得る	**CI（自動 Feedback）**
TDD は、高速な Feedback で設計と実装を小さく往復する有力な実践方法の一つ。必ず行う工程ではない	段ごとに得られる Evidence の種類が違う。上位互換ではない	SILS の次に PR が必然的に発生するわけではない

横断基盤｜8章	**Configuration Management / Reproducibility。**設計・検証・統合すべての Evidence が、どの構成に対するものだったかを後から特定できる状態を支える。最後に追加する工程ではない

講義では左から順に学んだが、実際の開発ではこの三つを並行して使う。前段をやらなければ後段が成立しない一本道ではない。

9.2 結論

ソフトウェアエンジニアリングとは、最初から完璧なソフトウェアを作る技術ではない。

変更を小さくし、変更の結果を高速に検証し、学習しながら安全に設計を改善し続けるための技術である。

> 軌道上では回せない開発サイクルを、地上で可能な限り高速に回しておく。
> そのために、ソフトウェアを分離し、試験可能にし、シミュレーション可能にし、自動化する。

今日までに作った検証と統合の道具を、チームの計画とインターフェース設計へ

WEEK 6｜前回
宇宙開発と システムズエンジニアリング

→

WEEK 7｜今日
ソフトウェア エンジニアリング

→

WEEK 8｜次回・講義最終回
プロジェクトマネジメント
依存を描く PERT 図とマージン。
コードを書く前のインターフェース定義

→

以降
作業期間・チーム開発、そして合宿へ

- マネジメントの本質はタスクの依存関係をほどくこと。依存を描く PERT 図と、計画を守るマージンを学ぶ
- インターフェースの二つ目の顔。Week 7 では変更を局所化し Testability を作る設計境界、Week 8 では複数人が並行開発するためのチーム間の契約として扱う
- 事前学習の講義は Week 8 で一区切り。以降は作業期間とチーム開発へ

Week 6 で考えた「初回打ち上げに必要な機能」をSWE視点で考え直す

Week 6 の宿題

模擬衛星の初回打ち上げに必要な機能を考える

→

同じ宿題を三つの回で考え直す

お題は Week 6 と同じで、

今日はソフトウェアエンジニアリングの考え方から

Week 6　SE｜初回に必要な機能は何か →

Week 7　SWE｜どう作り、どこで検証するか →

Week 8　PM｜誰といつ作るか

考える観点の例

- その機能のうち、Unit Test に落とせる判断はどこか（入力と期待を書けるか）
- SILS で確かめる範囲はどこか（制御ループとして回す必要があるか）
- CI に載せるものは何か。そして、実機でしか分からないことは何か