ソフトウェアの設計と実装について

1. 設計について
2. 実装について

1. 設計について

なぜ設計が重要なのか

- 適当に作るとグチャグチャになってしまい・・・
  - 読んでも何をしているか分からん　＝　可読性が低い
    - いつも（想定される範囲のどんな入力でも）正しく動くのか？
  - 複雑すぎてテストできない
    - 入出力のパターンが膨大に
  - 複雑すぎてデバッグできない
    - 追わなければならない情報が膨大に
  - 簡単に変更できない　→　保守性、拡張性が低い
    - 変えなければならない部分が膨大に

- 類似プログラムの開発効率を上げたい
  - コードを使い回したい
  - 設計を使い回したい

何が悪いのか

- グチャグチャって？（スパゲッティプログラム）
  - 手続きの複雑性
    - 実行されるルートを追うのが困難
  - データ構造の複雑性
    - どこから参照されるのか、どこを参照するのか
  - アルゴリズム（論理構造）の複雑性
    - 何と何が影響する、依存するのか

なぜ設計にコストをかけるのか

- 大人数で大規模なものを開発するため
  - 他人が簡単に理解できるようにする
  - 分担して部品を作れるようにする
- ソフトウェアの正しさを保証するため
  - 単体テスト、結合テスト
  - デバッグ時のトレーサビリティ向上
- 類似のソフトウェアや知識を再利用して安く早く作るため
  - ソフトウェア部品化（SPL）
  - デザインパターン

設計とは　＝　構造化

- 小規模なものなら正しく動くことがわかるし、意味が理解できる
  - 小さなブロックの組み合わせでプログラムを作る
  - ブロック同士の依存関係は少ないほうが良い

- 構造化：プログラムを「いい具合」に分割して考えること
- 設計：その「いい具合」を探求すること

- 広い意味では・・・
  - 要求定義
  - 機能設計
  - 構造設計 ←　これについて説明する
  - ふるまい設計
  - テスト設計

構造化プログラミング

- たとえばC言語（手続き型言語）
  - 構造化制御構文
  - 関数
- 「機能」を関数（手続き）単位で構造化
- データの構造化
  - 構造体
  - スコープの限定

「いい設計」の目指すところ

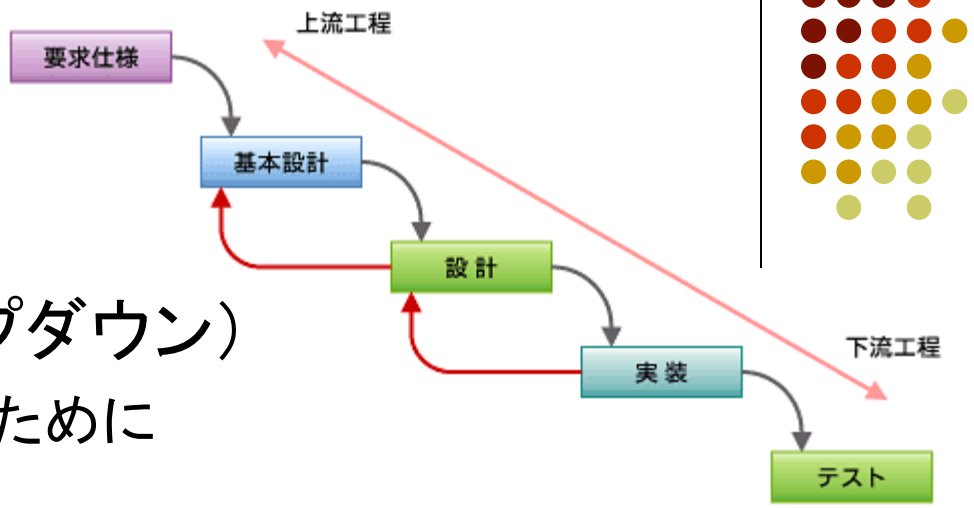
- あるクラスを作ったときに・・・
- 結合度＝他のクラスとの関連の強さ
  - なるべく低く。疎な結合にしたい。
  - 知らなくていいことは知らない
- 凝集度＝クラスの責務の専門性の高さ
  - なるだけ高く。専門バカを作る。

- 再利用性、変更容易性、可搬性、可読性を高く！

逆に悪い設計とは

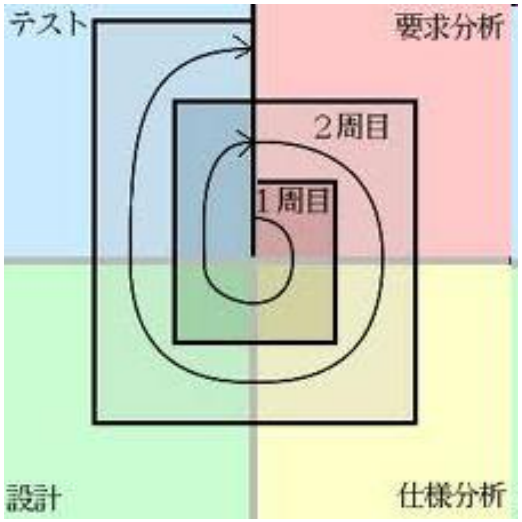
- 硬い
  - 変更しにくい。変更しようとすると他のたくさんの部分に変更が及ぶ
- もろい
  - 変更によって、その変更と概念的に関係ない部分が壊れる
- 移植性がない
  - 再利用できる部分をモジュールとして切り離すことが困難
- 不必要な繰り返し ←ループ（制御構造）のことではない
  - 同じような構造がたくさんある。抽象化できる部分がされていない
- 不透明
  - 読みにくく、分かりにくい

開発プロセス



- ウォーターフォール(トップダウン)
  - 手戻り(変更)を少なくするために
    上流設計を作り込む

- スパイラルモデル
  - 変更を許容するということ
  - 小さなループを回し、
    小規模なリリースを繰り返す
  - リファクタリングを続ける



ソフトウェアは生もの

- 開発中は常に変更にさらされる
  - 新しいアルゴリズムを思いついた
  - 実装してみたら要求を満たさなかった
- 行き当たりばったりの変更を続けていくと、
  ソフトウェアはどんどん腐っていく。
- リファクタリングを行う前提で進める
  - 機能・動作は変わらないが構造（設計）をより良くすること

アジャイル開発

- アジャイル＝すばしっこい　軽量開発とも
- 特徴
  - 短いイテレーション（反復）　１週間くらい
  - 各イテレーションで動くものをリリース
  - 仕様変更に伴って、リファクタリング
  - メンバー間のコミュニケーションを重視
- 適する問題領域
  - 10人以下の小規模なチームが1か所で開発
  - プロジェクト進行中に要求・仕様の変更が多い

テストファースト

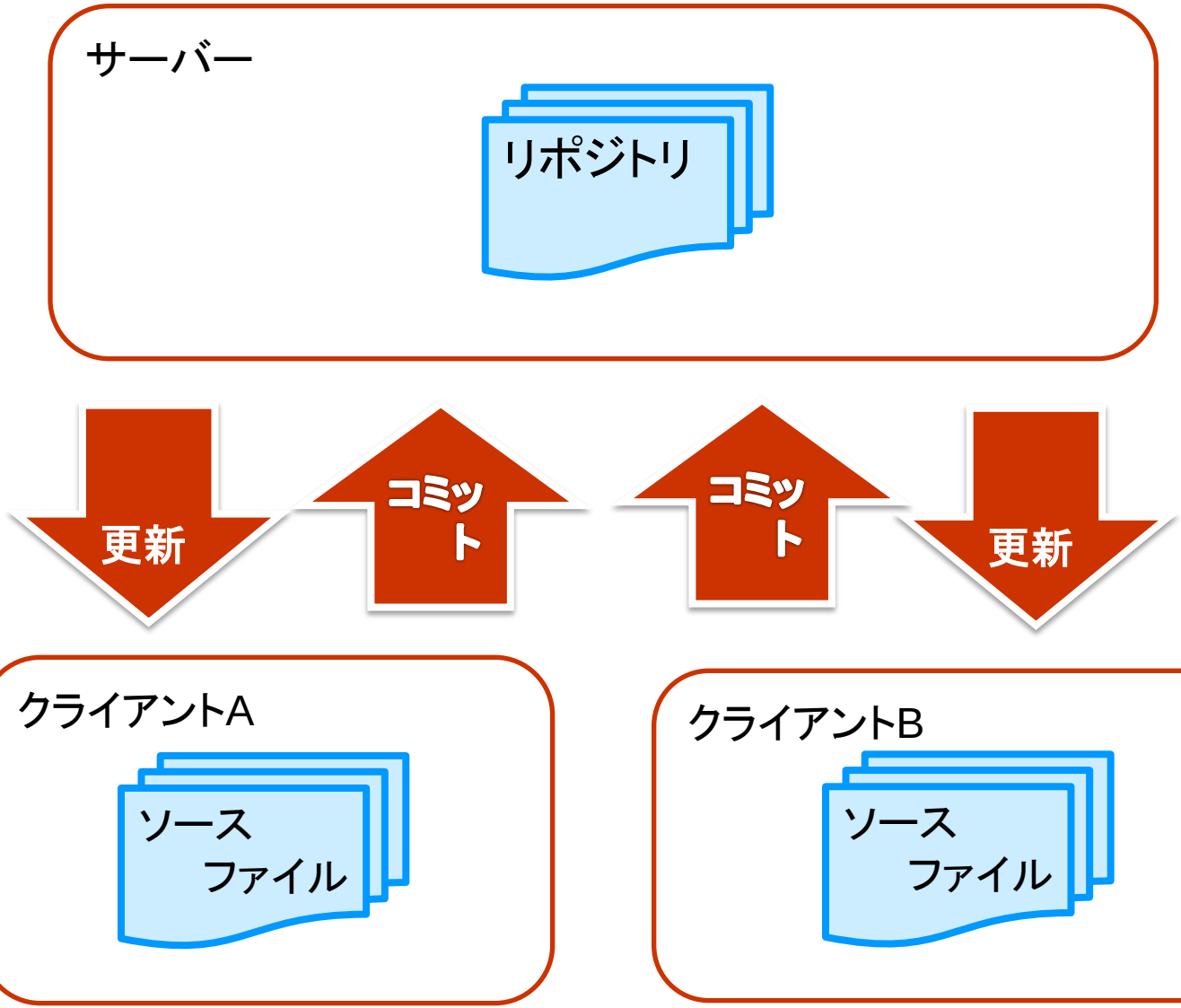
- どうせテストするんだから、先にテストを作る。
- テスト設計
  - テストケース
  - テスト用ダミーモック
  - テスタ（テストを実行するコード）
- 実装したらすぐテストをする
- テストに合格していない実装が存在しない
  - 本当はテストしていないコードをコミットすべきではないが、テスト設計を後回しにすると大体そういう事態になる

支援ツール紹介

- Tracによるプロジェクト管理
  - やること（Todo）の管理→チケット
  - ドキュメントの管理→Wiki

- Eclipse/Subversionによるリソース管理
  - 統合開発環境(IDE) Eclipse
  - リソース管理ツール　Subversion
    - 差分のコミット
    - ヒストリー追跡

Subversion



2. 実装について

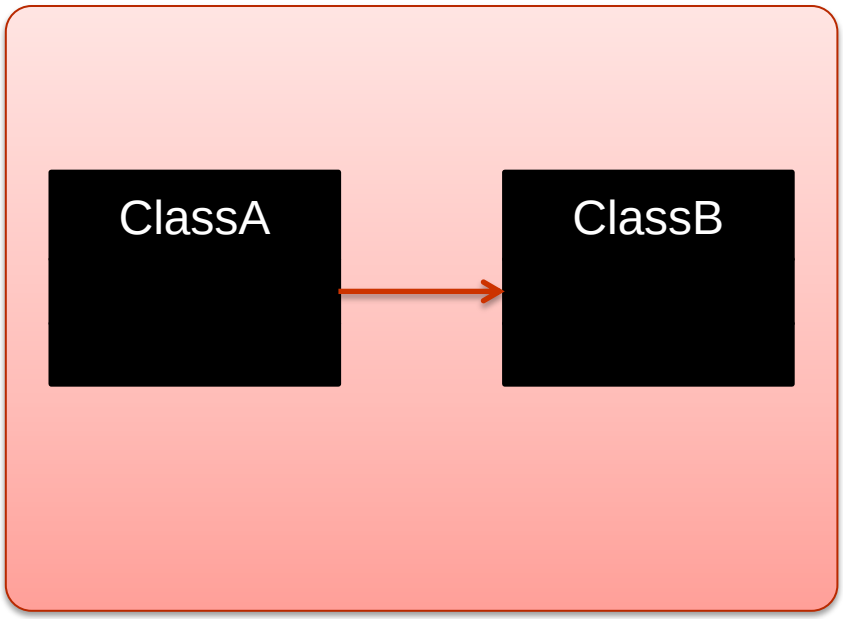
カプセル化とスコープ

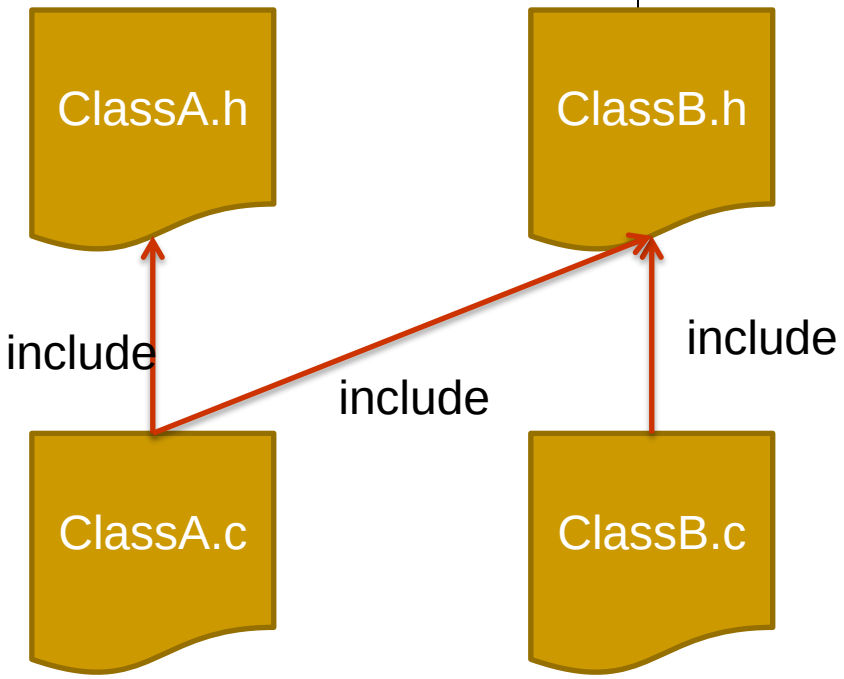
- カプセル化
  - 外部からは関係のない情報、実装を隠ぺいすることによってオブジェクトの独立性を高めること
    - グローバル変数、#define の使用を避ける
- 変数のスコープ
  - グローバル（リンク、実行ファイル単位）
  - ファイル単位（翻訳単位）
  - ブロック単位（{ブロック}、関数）
  - #defineのスコープは無限
    - 初期値付き変数で。static int PI =3.1415

ファイル分割について

- MyModule.h
  - 外部に公開する関数、変数のextern宣言
  - それらの使い方などのコメント
- MyModule.c
  - 依存する他のモジュールヘッダのinclude
  - 宣言
  - 実装の定義

- 他のファイルからは依存するモジュールのヘッダだけインクルードする





クラスごとに、実装(ソースファイル)　と　宣言(ヘッダーファイル)を作る。

インスタンスが一つしか生成されないクラスの実装方法

Color.h

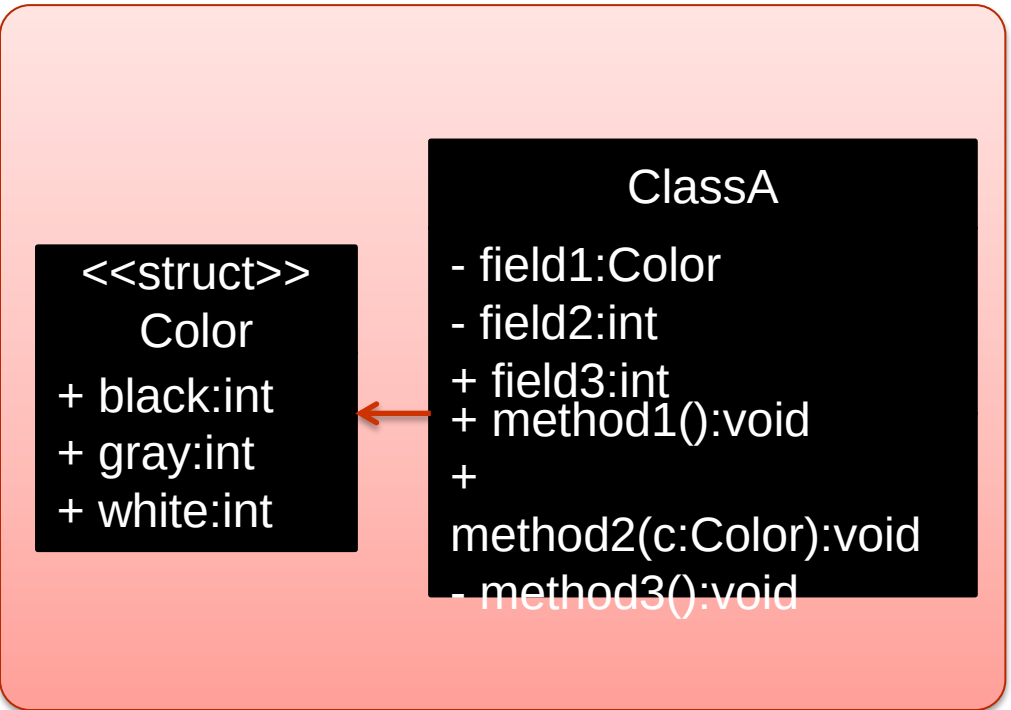
```
typedef struct {
  int black;
  int gray;
  int white;
} Color;
```

ClassA.h

```
include "Color.h"

exturn int ClassA_field3;

exturn void method1();
extern void method2(Color c);
```

ClassA.c

```
include "ClassA.h"

static Color field1;
static int field2;
int ClassA_field3;

void ClassA_method1(){ 実装 }
void ClassA_method2(Color c){ 実
装 }
void ClassA_method3(){
```



Stateパターン

- デザインパターン中の王道パターンでよく使われる
- オブジェクトの振る舞いが状態に依存するとき
- “状態”を抽象化してとらえたもの
- 状態をオブジェクトにする

- 状態の組み換えが容易
- 状態の追加や変更に強い

Stateの実装（ETロボコンにて）

```
91
92  case START:          //滑らかスタート
93      if(eod==ENTRY){
94          PIDCalculator_setSensor(VIRTUAL);
95          VirtualLineTracer_setSpeed( PIDCalculator_getGain()->normalForward );
96          VirtualLineTracer_setStraight();    //バーチャル直進開始
97          DetectBlack(W_SIDE);                //黒検知開始
98          LineTracer_setPosition(CENTER);     //検知後の位置
99          eod = DO;
100     }
101
102     VirtualLineTracer_do();
103     if( DetectBlack_checkEvent() == T){
104         state = CORRECT1; eod = ENTRY;
105     }
106     break;
107
108
109
110 case CORRECT1:
111     if(eod==ENTRY){
112         PIDCalculator_setSensor(REAL); //バーチャル終了
113         LineTracer_setEdge(LEFT);      //左エッジに変更
114         LineTracer_setRunState(RUN);
115         LineTracer_setSpeed( PIDCalculator_getGain()->normalForward ); //標準の前進速度
116
117         DetectDistance_absolute(2600);
118
119         eod = DO;
120     }
121
122     LineTracer_do();
123
124     if( DetectDistance_checkEvent() == T){
125         VirtualNXT_correction(1);//仮想補正ポイント１
126         ecrobot_sound_tone(440, 40, 100);
127         state = CORRECT2; eod = ENTRY;
128     }
129     break;
130
```

Entry処理

Do処理

遷移判定とExit処理

状態

状態

命名規則

- クラス名　LineTracer
- 変数　edgeChangeTime
- 定数　COURSE_WIDTH
- メソッド名　changeEdge()
  - 関数名としては LineTracer_changeEdge()

- 意味のある名前をつける（ループカウンタ変数はi,k,nなどを使ってよい）
- 他人が読んで分かるコードを書く
  - 関数内でも意味ごとのブロックに分けて構造化

コーディング

- コンパイルの通らないコードをコミットしない
  - 他の人がコンパイルする時に困る。
- コミットするまえに単体テストする。
  - 専用にテスト用のクラスを作る。
- コンパイルエラーは必ず読む
  - エラーの意味が分からなかったらググるor聞く
- Warningは必ずなくす
  - 危険な兆候を教えてくれる
    - ポインタと整数の比較
    - 初期化されずに参照される可能性のある変数

デバッグ心得

デバッグはバグの原因を探す作業ではない
”どうすれば原因を特定できるか”を考える作業である

- 必ず論理的に進めること
  - BlueToothロガー、ビープ音、画面表示などで
    - どこまで正常に実行されるかをチェックする
    - 前提としている入力は正しいのかチェックする