

要素技術とモデルを開発に使おう
要素技術をシステムに組込もう



ETロボコン実行委員会

3.0, 2020-06-29 11:13:54: 2020年用

目次

ねらいと目標	1
ねらい	1
目標	1
前提条件	2
準備	3
配布物	3
演習のゴール	4
ライントレーサーを動かしてみよう(sample03)	5
構造の図を描いてみる(クラス図)	6
クラス図にTracerクラスを追加する	7
ヘッダファイルから関連するクラスを追加する	8
Tracerクラスのメンバ関数とメンバ変数を追加する	10
Tracerクラスと他のクラスの関連を詳細化する	13
ライントレーサーの動作を確認する	15
クラス図にタスクと周期ハンドラを追加する	18
振舞いの図を描いてみる(アクティビティ図)	33
Tracerクラスにアクティビティ図を追加する	33
アクティビティ図に振舞いを追加する	34

走りの改善策を検討する(sample04)	38
sample03の走り方の課題はなにか	38
滑らかに走る方法を考えてみよう	39
プログラムで使えるよう整理しよう	40
うまくいきそうか確かめてみよう	41
比例制御を実験する	41
まだ実験しか終わっていなかった!	44
実験成果をモデルとコードに反映する(sample05)	45
実験成果を反映したモデル図を作成する	45
実験成果を反映したアクティビティ図	46
実験成果を反映したクラス図	47
モデル図に対応したコードを作成する	49
ここまでまとめ	53
要素技術をモデル図に表すには	54
要素技術を取り入れるときのポイント	54
要素技術をシステムに組込む手順	55
本資料について	56

ねらいと目標

ねらい

この章のねらいは、次の3つです。

- ・ モデルとコードが対応している状態を実感する
- ・ 要素技術が必要になったら、プログラムを作って実験してみる
- ・ 実験した結果を、モデル図とプログラムに反映する

目標

この章の目標は、次の5つです。

- ・ On/Off制御のライントレーサーを動かしてみる
- ・ サンプルのコードに対応しているモデル図を作成する
- ・ 動作の課題(もっと滑らかに走る)を検討する
- ・ 解決策(P制御)を実験して効果を確かめる
- ・ モデル図を更新して、実験結果をモデルとプログラムに反映する

前提条件

- ・「背景と準備」([text00](#))を修了していること
 - 「要素技術とモデルを開発に使おう」全体の目的、背景、学ぶことからを理解している
 - 開発環境が使える状態にある
- ・「コードとモデル図を対応づけてみよう」([text01](#))を修了していること
 - サンプルプログラムを動かすことができる
 - 基本的なEV3APIのクラス、メソッドを使ったことがある
 - 構造のモデル(クラス図、オブジェクト図)を描いたことがある

準備

- ・ 演習環境が整っていること
 - ETロボコン用のEV3開発環境が用意できていること
 - テキストエディタ(Visual Studio Codeを使います)が利用可能のこと
- ・ 基礎演習が済んでいること
 - Unixの基本的なコマンドを知っている
 - モデリングツールが用意できている(講師はAstahを使って演習します)
 - シミュレータの動作を確認できている
- ・ 配布物が揃っていること

配布物

- ・ テキスト(講師は `text02.pdf` を使います)
- ・ サンプルモデル図(`sample03.astah`)
- ・ サンプルコード(`sample03.zip`)
 - 自分の開発環境のワークスペースにコピーしておくこと
- ・ デモ動画ファイル(`sample03.mp4` 、 `sample05.mp4`)

演習のゴール

この章の演習では、On/Off制御のライントレーサーを、比例制御(P制御)によるライントレーサーにアップグレードします。演習する前に、動画を見てゴールをイメージしましょう。

- ・ On/Off制御のライントレーサー([sample03.mp4](#))
- ・ 比例制御によるライントレーサー([sample05.mp4](#))

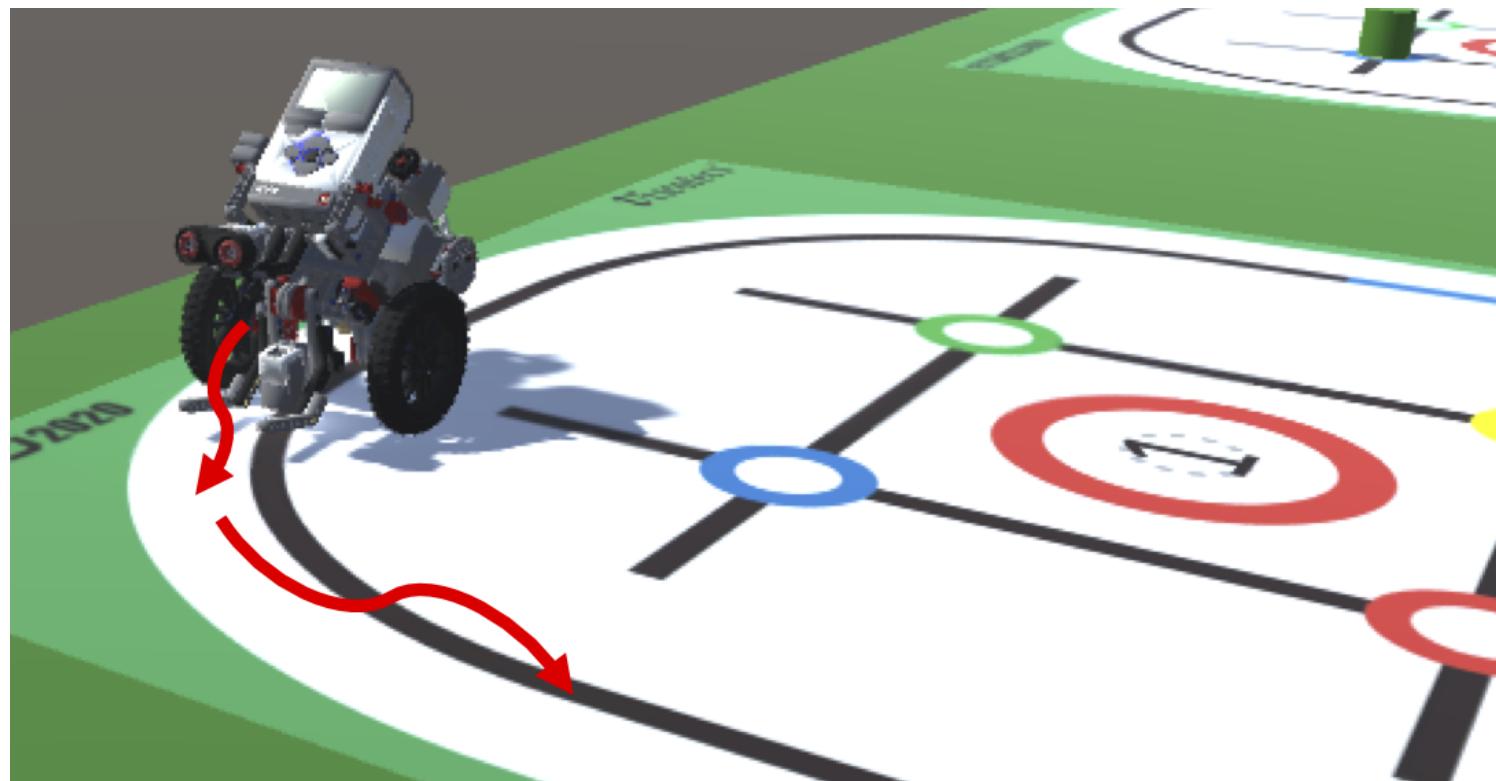


図 1. ライントレーサーが走行する様子

ライントレーサーを動かしてみよう(sample03)

On/Off制御を使った簡単なライントレースのサンプルを動かしてみましょう。

1. sample03 をビルドして、シミュレータ上で動かしてみましょう
 - `make app=sample03 sim up`
2. シミュレータが起動すると、走行体は反時計回りにラインの外周に沿って走行します
3. 左ボタンを長押しすると停止します
4. ライントレースがうまく行かないときは、カラーセンサーのしきい値(`mThreshold`)を調整してみましょう

みなさんも動かしてみましょう

ここで、みなさんも sample03 を動かしてみてください。動かしたあとで sample03 のモデル図を描く演習へ進みましょう。

構造の図を描いてみる(クラス図)

sample03 のコードを見ながら、コードの構造をクラス図で表してみましょう。

!

この演習は、意図的に、コードを読んで対応づいたモデル図を作るところから始めています。このような手順にしたのは、実装に使うモデルとコードの対応関係を学んでもらうためです。あえて「設計してから実装する」という開発プロセスには沿っていませんので注意しましょう。開発プロセスに沿った演習は、技術教育2「開発プロセスに沿って開発する」でやります。

先に説明を読み進め、その後で演習しましょう

先に sample03 のコードからクラス図を作るところまで説明します。
みなさんは、説明が終わってからクラス図を作成しましょう。

クラス図にTracerクラスを追加する

まず、クラス図を用意しましょう。

`Tracer.h` や `Tracer.cpp` は `app` ディレクトリに作成しています。このことがわかるように、モデルにも `app` パッケージを追加します。そして、`Tracer` クラスは、`app` パッケージの中に作ります(図 2)。



図 2. `app` パッケージを作り、その中に `Tracer` クラスを追加する

まだクラスがひとつだけですが、クラス図は 図 3 のようになるでしょう。

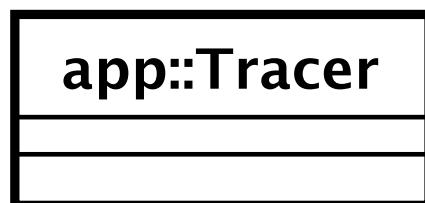


図 3. `sample03` のクラス図(1)

この段階では、まだクラスの詳細はわかりませんので、操作や属性は空欄です。名前空間をクラス名とともに明示しておくと、クラスの配置もわかるようになります。



クラス名に名前空間をつけるかつつけないかは、その図が示す範囲に応じて変わります。このクラス図では `sample03` 全体の構造を表そうと考えているので、図 2 の構造に合わせて書いておくのがよいでしょう。もし、`app` パッケージ自身のクラス図を描くのであれば、必ずしも名前空間を書かなくてもよいでしょう。

ヘッダファイルから関連するクラスを追加する

`Tracer` クラスが用意できたので、ヘッダファイル `Tracer.h` に書いてあることを反映してみましょう。最初は、インクルードしているヘッダファイルの情報に着目します（リスト 1）。

リスト 1. `sample03/app/Tracer.h`

```
1 #include "Motor.h"          ①
2 #include "ColorSensor.h"    ②
3 #include "util.h"           ③
4
5 using namespace ev3api;   ④
```

- ① Motor クラスを使うためのヘッダファイルをインクルードした
- ② ColorSensor クラスを使うためのヘッダファイルをインクルードした
- ③ LCDに情報を表示するユーティリティ関数を使うために util.h をインクルードした
- ④ 名前空間 ev3api を使う宣言(APIを呼び出す時に空間名を省略できる)

util.h は演習用のユーティリティ関数を宣言したヘッダファイルなので、この演習で図に反映する対象から外しておきます。

Motor.h と ColorSensor.h は、EV3RTの ev3api ライブラリの提供です。「EV3RT C++ API Reference」で調べると、それぞれ Motor クラスと ColorSensor クラスを提供しているのがわかります。モデル図にもEV3APIパッケージを追加して、そこにクラスを追加しましょう。

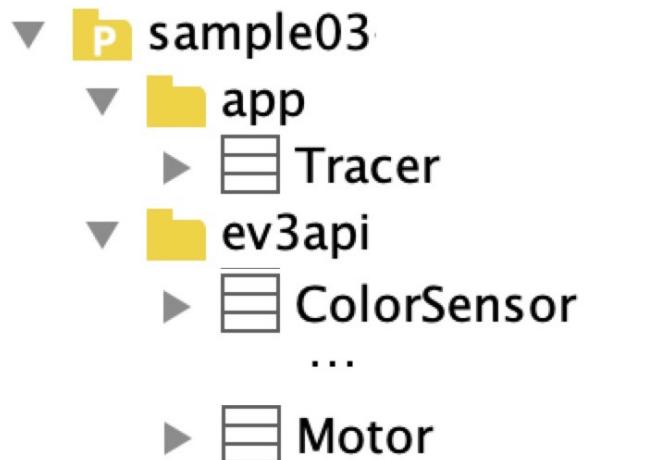


図 4. EV3API パッケージを作り、その中に Motor 、ColorSensor クラスを追加する

そして、クラス図に `Motor` クラスと `ColorSensor` クラスを追加しましよう(図 5)。EV3APIのクラスであることがわかるよう、これらのクラスにも名前空間をつけておくとよいですね。この図では、`Motor`、`ColorSensor` クラス属性と操作は省略(非表示に)しています。最後に、`Tracer` クラスが `Motor` クラスと `ColorSensor` クラスをインクルードして利用していることがわかるよう、関連を引いておきます。

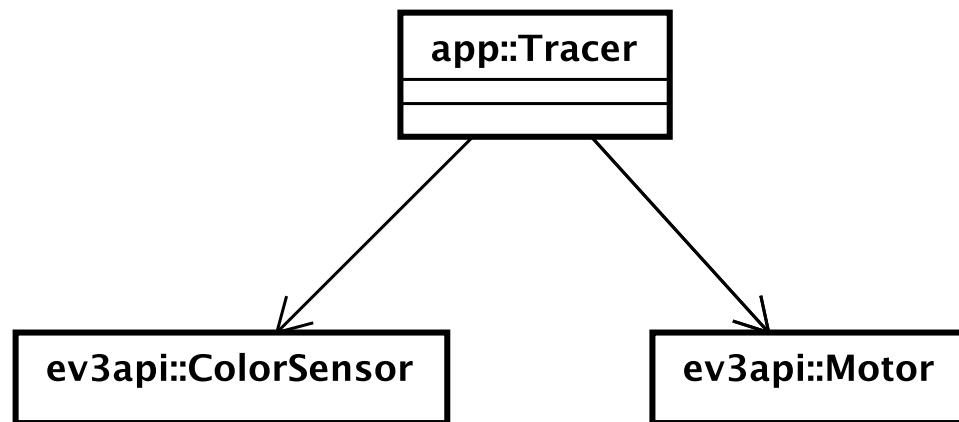


図 5. sample03 のクラス図(2)

Tracerクラスのメンバ関数とメンバ変数を追加する

引き続き `Tracer.h` を読み進めます。こんどはクラス宣言を見てみましょう(リスト 2)。

リスト2. sample03/app/Tracer.h (つづき)

```
7 class Tracer { ①
8 public:
9   Tracer();
10  void run();      ②
11  void init();
12  void terminate();
13
14 private:
15  Motor leftWheel;
16  Motor rightWheel;
17  ColorSensor colorSensor; ③
18  const int8_t mThreshold = 20; ④
19  const int8_t pwm = (Motor::PWM_MAX) / 6;
20 };
```

- ① Tracer クラスの宣言の始まり
- ② メンバ関数 run (走行する) 、init (初期化する) 、terminate (終了処理) の宣言
- ③ ColorSensor クラスを使ったメンバ変数の宣言
- ④ メンバ変数 colorSensor を評価するとき「しきい値」に使う定数の定義

コンストラクタの他に、`public` なメンバ関数として、`run`、`init`、`terminate` を宣言しています。これらを `Tracer` クラスの操作に追加します(図 6)。あとに続く `private` なメンバ変数のうち、`mThreshold` と `pwm` は、いずれも初期値のある定数になっています。これらを属性に追加し、定数と分かれるよう、`static` を表す下線と初期値を書いておきましょう。

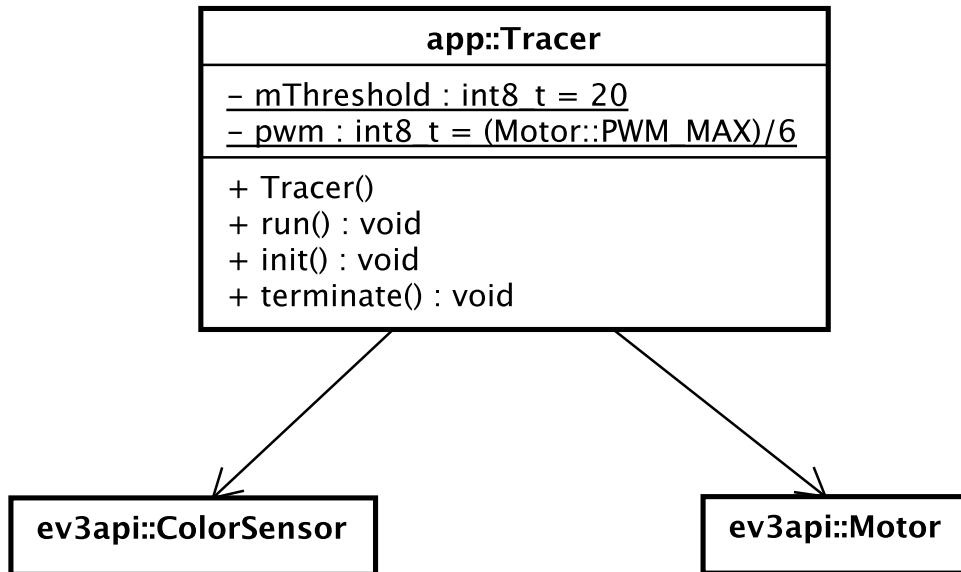


図 6. sample03 のクラス図(3)

Tracerクラスと他のクラスの関連を詳細化する

残るメンバ変数は、`leftWheel` と `rightWheel`、そして `colorSensor` です。メンバ変数が他のクラスのインスタンス(あるいはインスタンスへの参照やポインタ)を表すときは、属性を使う代わりに関連を使って表します。

図6では、`Tracer` が `leftWheel` と `rightWheel` を区別して利用していることが表せていません。そこで、`Tracer` クラスから、`Motor` クラスへの関連をもう少し詳細化します(図7)。

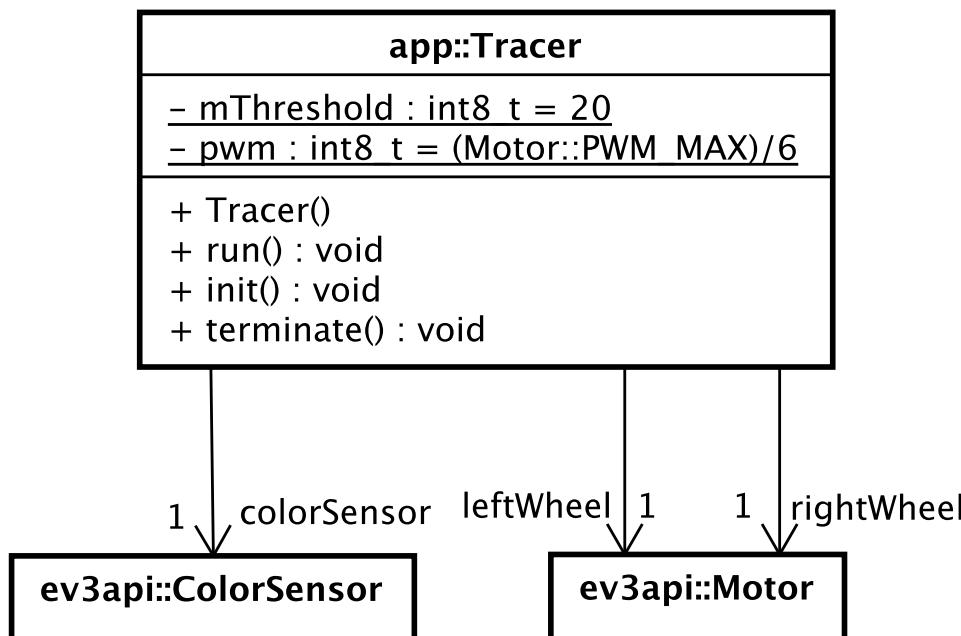


図7. sample03 のクラス図(4)

別々に2つの役割があることを示すため、関連を2本に分けました。そして、関連端名として `leftWheel` と `rightWheel` を追加しました。メンバ変数 `colorSensor` についても同じように関連端名を追加しました。

関連端名で示している3つのインスタンスは、`Tracer` が保有し、生成と破棄の責務を担っています。このような関係を表す関連を「コンポジション」と呼び、所有するクラス側の関連端に黒塗りのダイアモンドで表します(図8)。

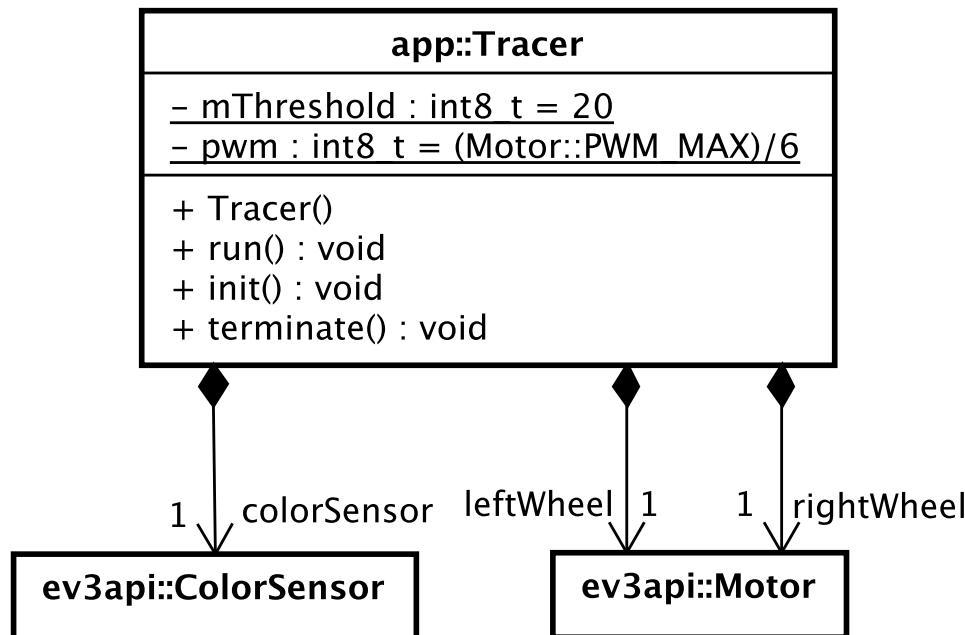


図 8. sample03 のクラス図(5)

みなさんもクラス図を描いてみましょう

みなさんも、[sample03](#) のコードを見ながらクラス図を描いてみましょう。同じような図が描けたら、いったん保存して、次の説明へ進みましょう。

ライントレーサーの動作を確認する

こんどは、[Tracer.cpp](#) を見てみましょう。

リスト 3. [sample03/app/Tracer.cpp](#)

```
1 #include "Tracer.h" ①
2
3 Tracer::Tracer():
4     leftWheel(PORT_C), rightWheel(PORT_B), colorSensor(PORT_3) { ②
5 }
6
7 void Tracer::init() {
8     init_f("Tracer");
9 }
```

- ① Tracer クラスを宣言したヘッダファイルをインクルードした
- ② コンストラクタの初期化リストで、メンバ変数をそれぞれ初期化した

i | コンストラクタの初期化リストについては、技術教育2の資料に説明があります。

リスト4. sample03/app/Tracer.cpp (つづき)

```
7 void Tracer::init() {  
8     init_f("Tracer");  
9 }  
10  
11 void Tracer::terminate() {  
12     msg_f("Stopped.", 1);  
13     leftWheel.stop(); ①  
14     rightWheel.stop();  
15 }
```

- ① 終了時に両輪を停止する

ここまででは、あまり難しいことはないでしょう。

では、run メソッド(リスト5)を見てみましょう。

リスト 5. sample03/app/Tracer.cpp (抜粋)

```
17 void Tracer::run() {  
18     msg_f("running...", 1);  
19     if(colorSensor.getBrightness() >= mThreshold) { ①  
20         leftWheel.setPWM(0);  
21         rightWheel.setPWM(pwm);  
22     } else { ②  
23         leftWheel.setPWM(pwm);  
24         rightWheel.setPWM(0);  
25     }  
26 }
```

① 明るい時(ライン外)の処理。左に曲がっている

② 暗い時(ライン上)の処理。右に曲がっている

ライン上かラインから外れているかによって、走行体の左右の車輪のどちらか一方だけ回しています。`run` メソッドは、あとで説明する周期ハンドラが繰り返し呼び出すので、この動作が繰り返されます。その結果、走行体は経路に沿って走るというわけです。

この振舞いを図に表す演習は、「振舞いの図を描いてみる(アクティビティ図)」でやります。

クラス図にタスクと周期ハンドラを追加する

こんどは、`app.cpp` の処理をみて、クラス図に反映しましょう。

!

この演習では、RTOS(リアルタイムオペレーティングシステム)やタスク、μITORNのタスクコンフィギュレーションについての説明を省きます。

これらを説明することは、演習の目的ではないし、むしろRTOSの講義の方が多くなってしまうからです。まだよく知らなくても、いまはあまり気にしないで演習を進め、詳しいことはあとで調べるようにしましょう。

リスト 6. `sample03/app.cpp`

```
1 #include "app.h" ①
2 #include "Tracer.h" ②
3 #include "Clock.h" ③
4 using namespace ev3api;
5
6 Tracer tracer; ④
7 Clock clock; ⑤
```

- ① タスクや周期ハンドラの宣言をインクルードした

- ② Tracer クラスの宣言をインクルードした
- ③ Clock クラスの宣言をインクルードした
- ④ Tracer クラスの静的なインスタンス `tracer` を定義した
- ⑤ Clock クラスの静的なインスタンス `clock` を定義した

`tracer` と `clock` は、大域的(グローバル)で静的なインスタンスです。静的なインスタンスは、アプリケーションの初期化時に作成され、アプリケーションが終了するとき破棄されます。

sample03 のクラス図に `Clock` クラスを追加しましょう(図 9)。名前空間もつけておきましょう。

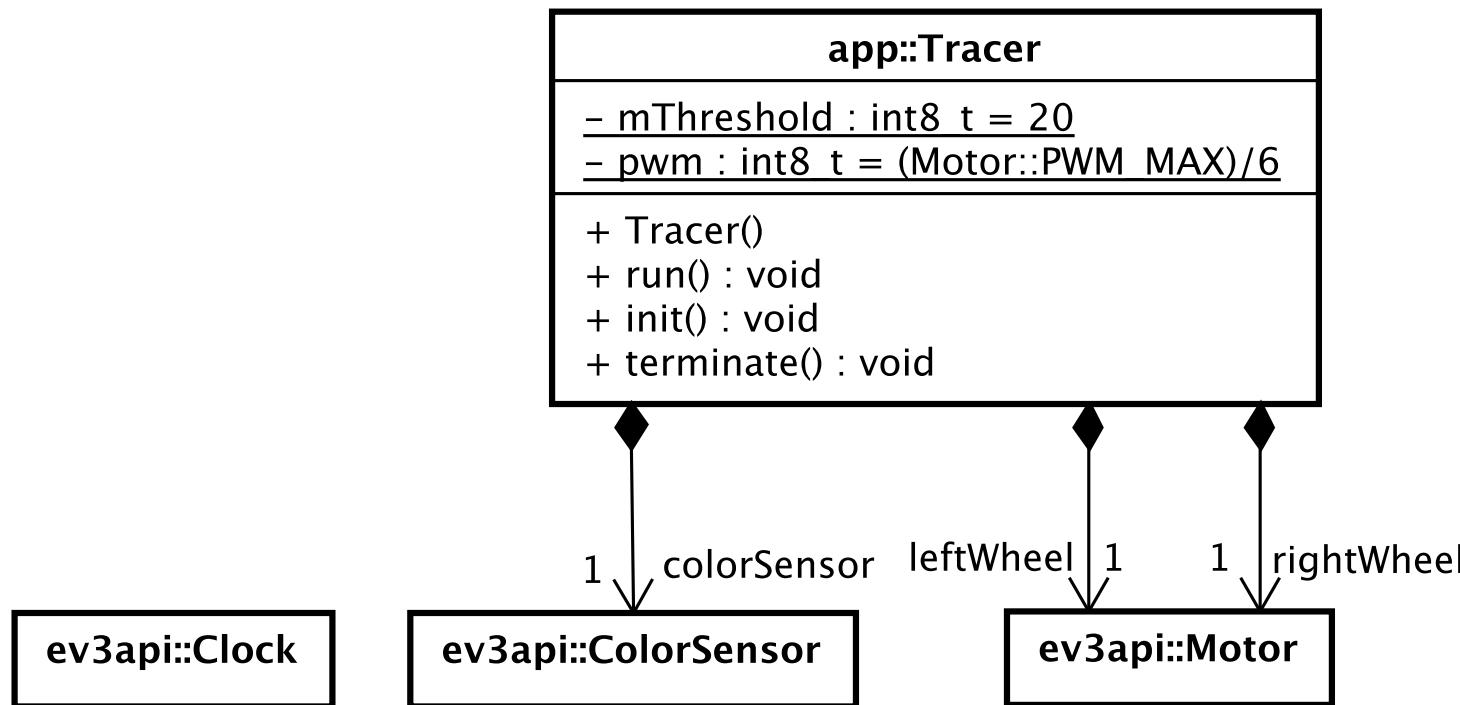


図 9. sample03 のクラス図(6)

つづきを見てみましょう。

リスト 7. sample03/app.cpp (つづき)

```
9 void tracer_task(intptr_t exinf) { ①  
10   tracer.run(); ②  
11   ext_tsk();  
12 }
```

① TRACER_TASK のメインルーチン tracer_task の定義

② tracer の run メソッドを呼び出す

tracer_task は、ラインをトレースする処理を担当するタスク TRACER_TASK のメインルーチンです。メインルーチンは、OSから呼び出されて動作する特別な関数です。



タスクについての簡単な説明は、【コラム】タスクについて を参照してください。
また、詳細な仕様は、「TOPPERS新世代カーネル統合仕様書」を参照してください。

【コラム】タスクについて

タスクは、プログラムを実行する単位です。詳細は異なりますが、スレッド、プロセスと似た考え方です。RTOS上のシステムが、同時に複数の処理を並行に動作させたい場合は、複数のタスクを使えます。複数のタスクを使う方法を「マルチタスク」と呼びます。

たとえば、`Clock` クラスの `sleep` メソッドを使って時間の経過を待つとします。しかしこの方法では、プログラムが時間の経過を待っている間、ボタンが押されるのを監視するといった他の処理はできません。そこで、別のタスクを用意して、ボタンを監視する処理を並行して実行します。

μ ITRONには、タスクを作る方法に、コンフィギュレーションファイルにおいて静的API `CRE_TSK` を使う方法と、プログラム中からサービスコールの `cre_tsk` を使う方法があります。

`CRE_TSK` の構文とパラメータの意味については、TOPPERS/EV3RTのWebページ「[開発環境の構築に関する質問](#)」の「タスクの優先度を変更するには、どうすれば良いでしょうか？」の説明を参照しましょう。

`sample03` のタスクの定義は、コンフィギュレーションファイル `app.cfg` に書いてあります。設定を確認してみましょう（リスト 8）。

リスト 8. sample03/app.cfg (抜粋)

```
5 DOMAIN(TDOM_APP) {  
6 CRE_TSK( MAIN_TASK,  
7   { TA_ACT, 0, main_task, MAIN_PRIORITY, STACK_SIZE, NULL } ); ①  
8 CRE_TSK( TRACER_TASK,  
9   { TA_NULL, 0, tracer_task, TRACER_PRIORITY, STACK_SIZE, NULL } ); ②  
10  
11 CRE_CYC( TRACER_CYC,  
12   { TA_NULL, { TNFY_ACTTSK, TRACER_TASK}, 50*1000, 1*1000}); ③  
13 }
```

- ① タスク MAIN_TASK を生成し、main_task を動かす
- ② タスク TRACER_TASK を生成し、tracer_task を動かす
- ③ 周期ハンドラ TRACER_CYC を生成し、TRACER_TASK を周期的に起動する

MAIN_TASK は、プログラム起動時から動作します (TA_ACT)。TRACER_TASK は、プログラム中で起動されるまで待機します (TA_NULL)。周期ハンドラ TRACER_CYC も、プログラム中で起動されるまで待機します (TA_NULL)。TRACER_CYC は動作している間、TRACER_TASK を $50*1000 \mu\text{s}$ (50ms) ごとに繰り返し起動します。この結果、tracer_task の内部に繰り返し処理がなくても、あたかも繰り返し処理しているかのように振る舞います。



コンフィギュレーションファイルについての簡単な説明は、【コラム】コンフィギュレーションファイルについてを参照してください。

また、詳細な仕様は、「TOPPERS新世代カーネル用コンフィギュレータ仕様書」を参照してください。



カラーセンサーは、計測に 1ms 程度の時間が必要です。それより短い周期で測定すると、期待した動作にならないことがありますので、注意しましょう。

【コラム】コンフィギュレーションファイルについて

組込み機器は、電源を入れるとそのまま機器の提供する機能の動作へと進みます（プログラム選んだりしませんよね）。組込み機器がそのように動作するのは、機能が動き出す前に、デバイスを初期化や機器の機能の設定を済ませておくからです。

コンフィギュレーションファイルは、このような設定を記述しておくファイルです。 μ ITRONの場合も、タスクや周期ハンドラを生成するときの設定などに使います。そして、コマンド（コンフィギュレータ）を使って、OSが参照する設定ファイルやアプリケーションが使うヘッダファイルに変換します。



周期ハンドラについての簡単な説明は、【コラム】周期ハンドラを参照してください。また、詳細な仕様は、「TOPPERS新世代カーネル統合仕様書」を参照してください。

【コラム】周期ハンドラ

システムの開発では、一定時間ごとに処理を繰り返したい場合がよくあります。このような場合、以前はタイマー割込みを使うのが一般的でした。タイマー割込みは重要な技術ですが、実装環境に依存しやすく、取り扱いが難しい技術もあります。そこで、μITRONでは、より安全に使用できる周期処理を用意したいと考えて、周期ハンドラを提供しています。

コンフィギュレーションファイルで周期ハンドラの生成を設定するには、`CRE_CYC` を使います。周期ハンドラが実行するメインルーチン(関数ポインタ、関数名を書く)には、ハンドラとして動作するCの関数を指定します。指定する関数は、必ずしもタスクである必要はありませんが、タスクを起動したいときは、`TNFY_ACTTSK` を指定します。

`CRE_CYC` の構文とパラメータの意味については、TOPPERS/EV3RTのWebサイトの「開発環境の構築に関する質問」の「周期的な処理を追加するためには、どうすれば良いでしょうか?」の説明を参照しましょう。



周期ハンドラを使ったプログラムの構成については、技術教育2の資料に説明があります。

この演習では、タスクと周期ハンドラをクラスとして扱ってみましょう。タスクにはステレオタイプ `task` を、周期ハンドラにはステレオタイプ `cyclic_handler` をつけて、通常のクラスと区別して表すことに決めます。

タスクのメインルーチンは、タスクが提供している操作とみなします。そして、通常のクラスとは異なる要素をクラスや操作で表現したことを、ノートをつけて補足することに決めます。



UMLは、タスクや周期ハンドラをそのまま表現する記法を提供していません。代わりに「アクティブクラス」を表す記法を提供しています。アクティブクラスを扱えるモデリングツールの場合は、使ってみるとよいでしょう。

これで、タスクや周期ハンドラを表現できるようになりましたので、`TRACER_TASK`、`TRACER_CYC`をクラス図に追加しましょう(図 10)。内部の処理で `tracer` を参照しているので、`Tracer` クラスとの間に関連と関連端名も追加しました。

タスクと周期ハンドラは、クラスを用いて表し、ステレオタイプで区別している。
また、タスクのメインルーチンをクラスの操作を使って表している。

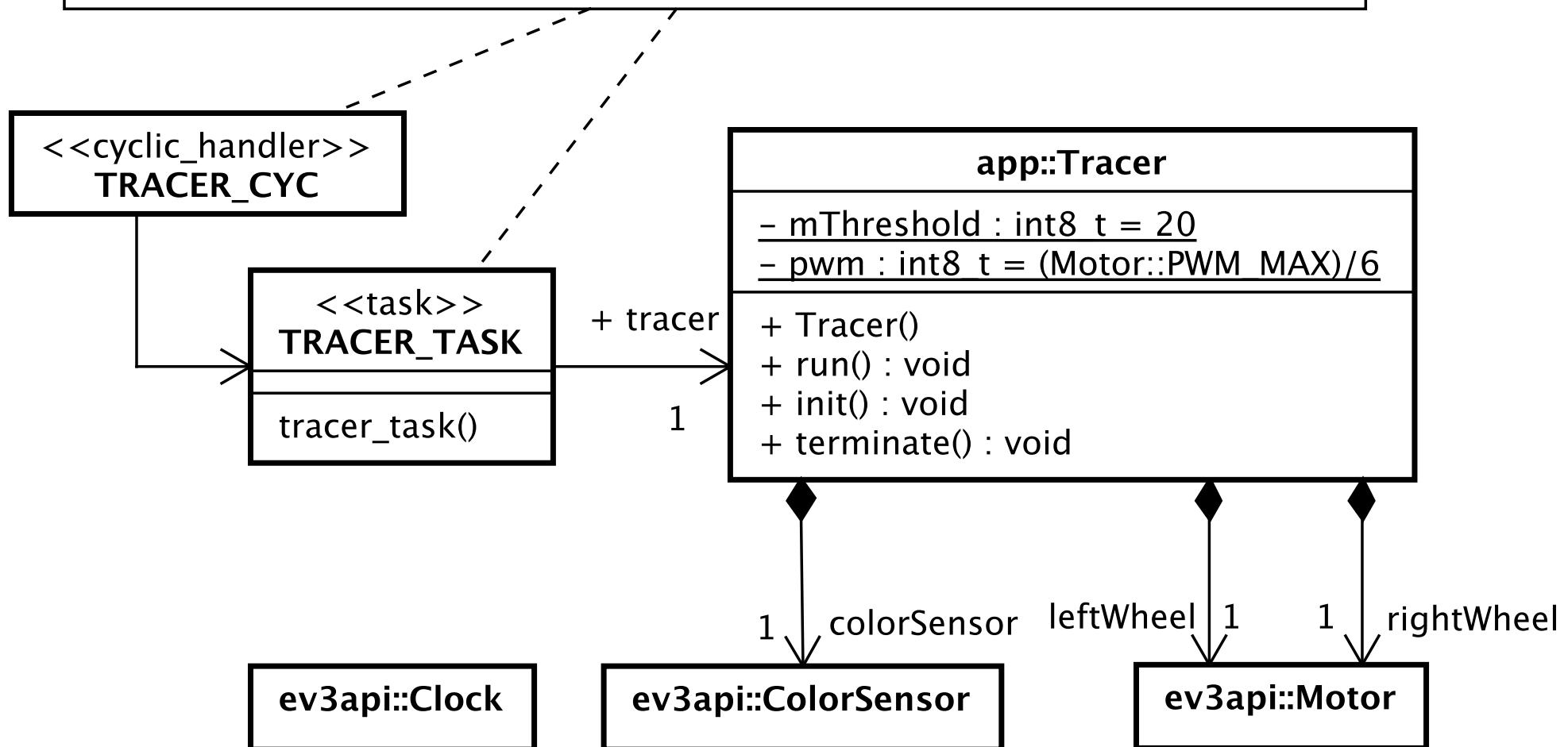


図 10. sample03 のクラス図(7)

つづいて、`main_task` の処理も見てみましょう。

リスト 9. sample03/app.cpp (つづき)

```
14 void main_task(intptr_t unused) { ①
15   const uint32_t duration = 100; ②
16
17   tracer.init(); ③
18   sta_cyc(TRACER_CYC); ④
```

- ① タスク `main_task` の定義
- ② スリープするときの時間(ミリ秒)の設定
- ③ `tracer` の初期化処理
- ④ 周期ハンドラ `TRACER_CYC` を起動して、これ以降周期的に `TRACER_TASK` が起動する

`sta_cyc` を使って周期ハンドラ `TRACER_CYC` を起動すると、`tacer_task` が周期的に呼び出されるようになります。

`MAIN_TASK` と `main_task` もクラス図に追加しましょう(図 11)。

ステレオタイプ `task` の場合、メインルーチン(ここでは `main_task`)の変数、定数は属性に書いておくことにします。このことも、ノートで補足しておきます。

main_task の場合は、初期値のある定数 duration を属性に追加しておきましょう。

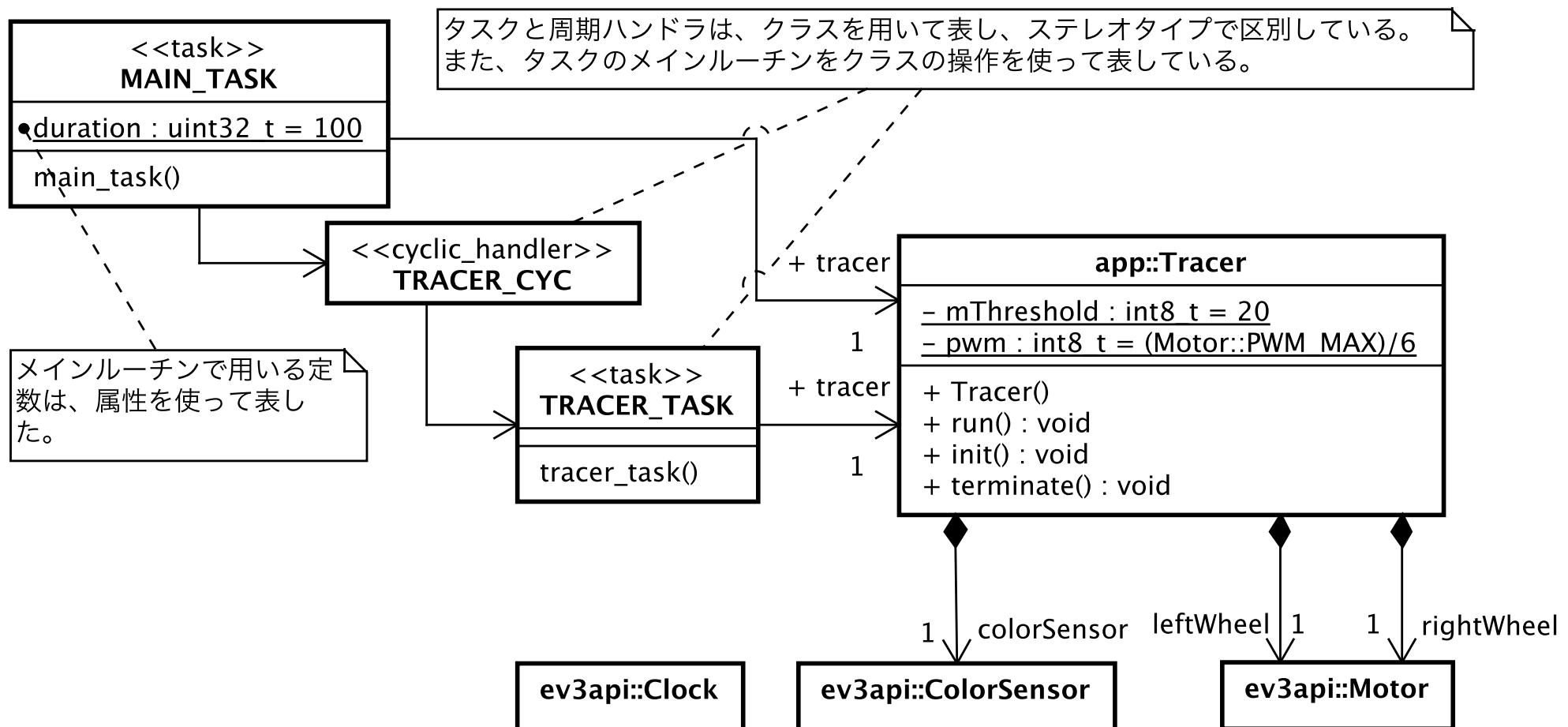


図 11. sample03 のクラス図(8)

残りの処理も見てみましょう。

リスト 10. sample03/app.cpp (つづき)

```
20 while (!ev3_button_is_pressed(LEFT_BUTTON)) { ①
21     clock.sleep(duration); ②
22 }
23
24 stp_cyc(TRACER_CYC); ③
25 tracer.terminate(); ④
26 ext_tsk(); ⑤
27 }
```

- ① EV3本体のボタンが押されるまで繰り返す
- ② `duration` で指定した時間スリープする
- ③ 周期ハンドラ `TRACER_CYC` を停止する
- ④ `tracer` の終了処理
- ⑤ 自タスク(`MAIN_TASK`)を休止状態へ移行する

`duration` で指定した時間スリープしては、左ボタンが押されているか調べています。この間、別のタスク `TRACER_TASK` が周期ハンドラ `TRACER_CYC` によって定期的に起動を繰り返して、ラ

インをトレースしています。ボタンが押されたのを検知したら、周期ハンドラを停止して、`tracer` の終了処理を呼び出して処理を終了しています。

`main_task` が `clock` を使っていることがわかるよう、クラス図に関連を追加します(図 12)。

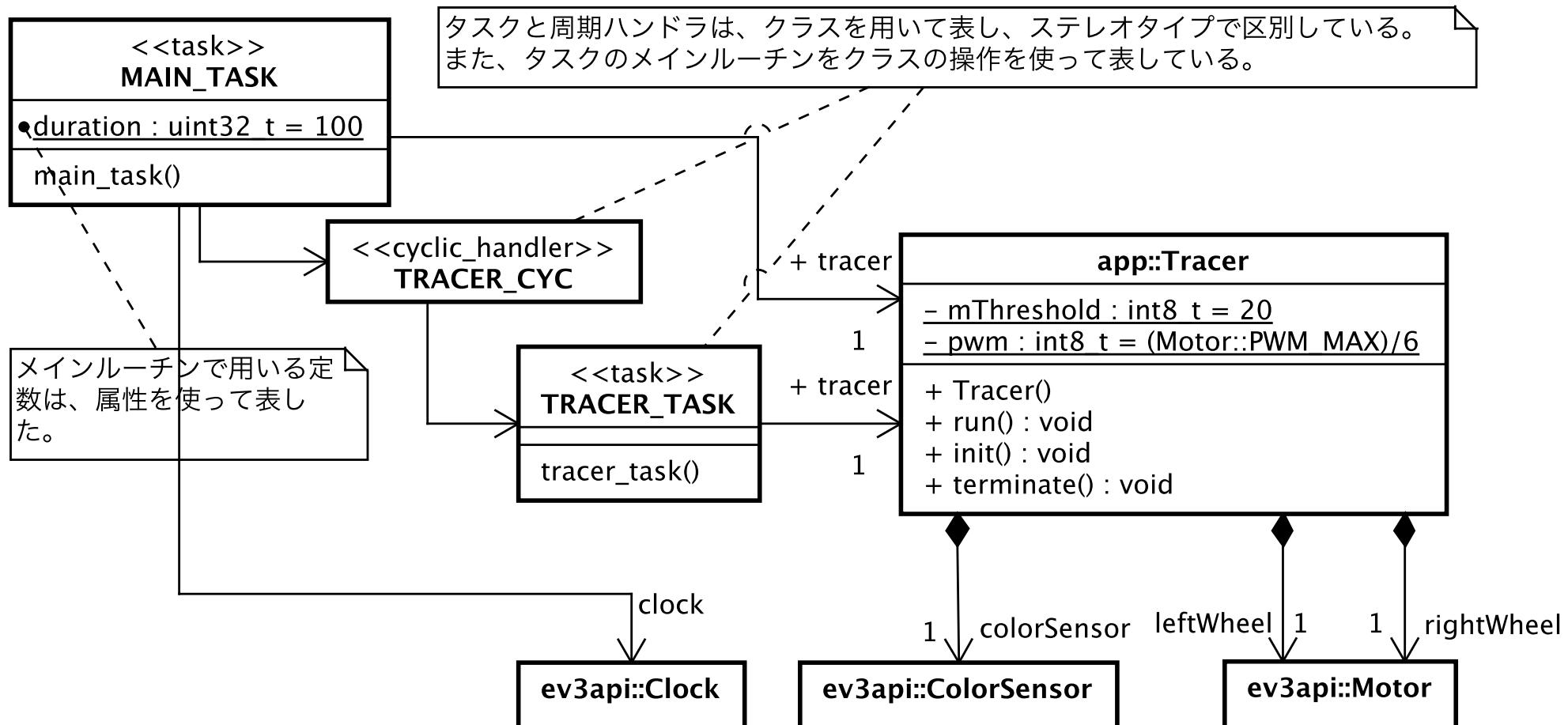


図 12. sample03 のクラス図(9)

これで、sample03 のプログラム全体の構造をクラス図に表すことができました。

設計から実装へ進むときは、この演習の逆の手順を使います。設計する前に、どのようにクラス図を描いたら、どのような実装にするのか決めます。たとえば、タスクをクラスに見立てたように、対応づけを先に決めておきます。設計するときは、その対応づけを使うことを前提に図を描きます。

みなさんもクラス図を更新しましょう

みなさんも、sample03 のコードを見ながらクラス図を更新してみましょう。同じような図が描けたら、いったん保存して、次の説明へ進みましょう。

振舞いの図を描いてみる(アクティビティ図)

クラス図は構造を表す図なので、どのように走行するかという振舞いについては表せません。振舞いを表すには、アクティビティ図やステートマシン図、あるいはシーケンス図を使います。

Tracerクラスにアクティビティ図を追加する

処理の流れを描くときには、アクティビティ図をよく使います。ここでは、Tracer クラスの run メソッドの処理の流れをアクティビティ図で表してみましょう。あまり複雑なメソッドではないので、アクティビティ図の練習にちょうどよいでしょう。

まず、アクティビティ図を追加しましょう。

ここで描こうとしているアクティビティ図は Tracer クラスの run メソッドの振る舞いを表すものです。そこで、Tracer クラスに関するモデルのひとつとして追加します(図 13)。



図 13. Tracer クラスに run メソッドのアクティビティ図を追加する

アクティビティ図に振舞いを追加する

もう一度、Tracer クラスの run メソッドを示しておきます(リスト 11)。

リスト 11. sample03/app/Tracer.cpp (抜粋)

```
17 void Tracer::run() {  
18     msg_f("running...", 1);  
19     if(colorSensor.getBrightness() >= mThreshold) { ①  
20         leftWheel.setPWM(0);  
21         rightWheel.setPWM(pwm);  
22     } else { ②  
23         leftWheel.setPWM(pwm);  
24         rightWheel.setPWM(0);  
25     }  
26 }
```

① 明るい時(ライン外)の処理。左に曲がっている

② 暗い時(ライン上)の処理。右に曲がっている

この run メソッドの処理の流れをアクティビティ図で表すと、図 14 のようになるでしょう。

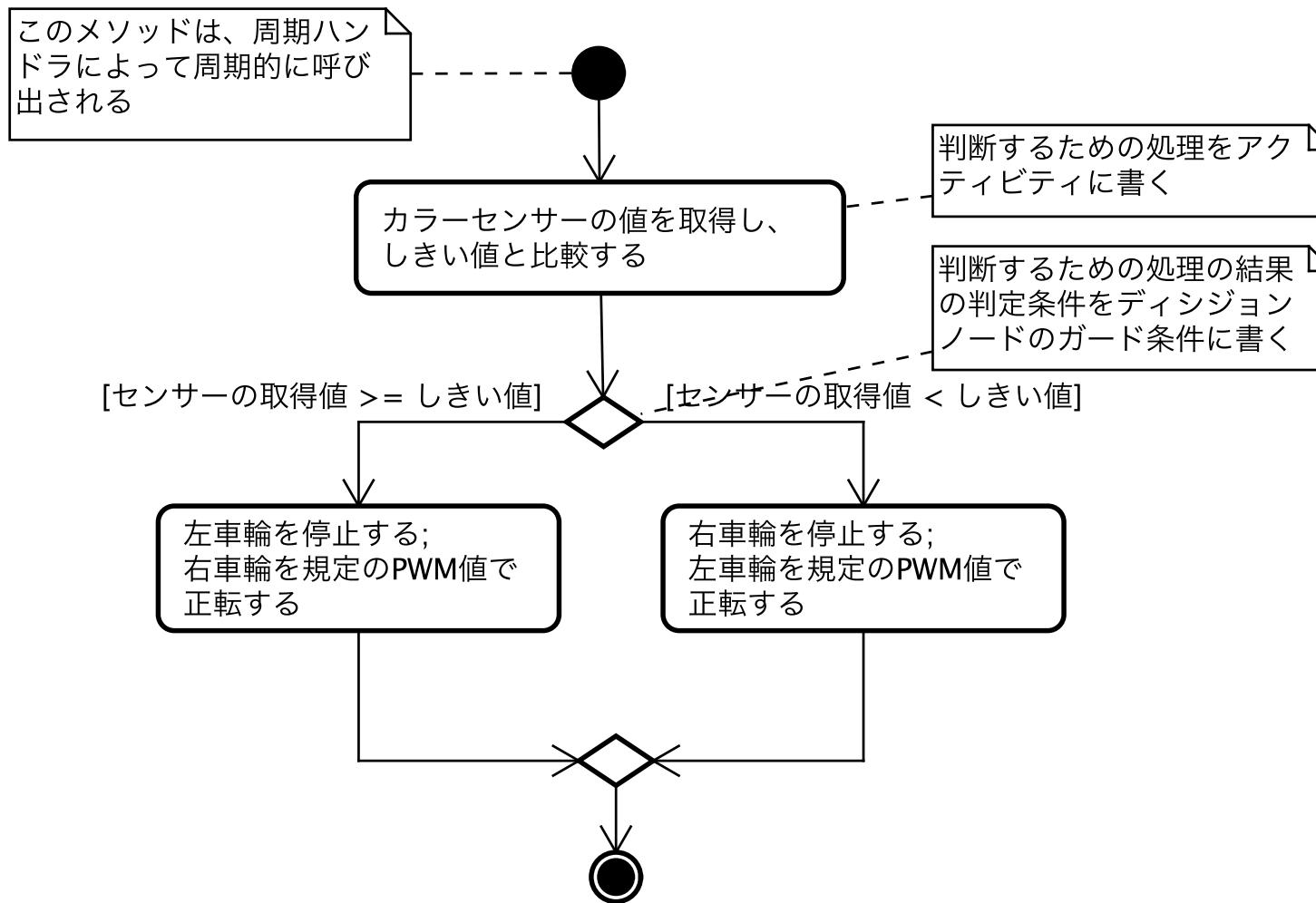


図 14. Tracer クラスの run メソッドのアクティビティ図

メソッドのアクティビティ図を描くポイントは次の通りです。

- ・「開始ノード」はひとつにする
- ・「終了ノード」のある図にする(繰り返しから必ず抜ける)
- ・処理には「アクションノード(アクティビティ)」を使う
- ・条件分岐には「ディシジョンノード」を使う
 - 条件分岐の判定処理には「アクションノード」を使う
 - 判定処理による分岐条件は「ガード条件」で表す
- ・メソッドの処理の流れを表すだけなら、パーティションは必ずしも必要ない
 - 他のクラスとのやり取りを表したいときは、シーケンス図を使う

あまり処理が長く複雑になると、アクティビティ図が描きにくくなってしまいます。そのときは、複数のメソッドに処理を分けて、その組み合わせで再構成したほうがよいでしょう。



この演習は、コードと対応づいている図を描くのが目標ですから、コードの処理の意味をあえて解釈していないことに注意しましょう。
モデル図と処理内容の対応づけについては、【コラム】アクティビティと実装を対応づけるを参考してください。

みなさんもアクティビティ図を描いてみましょう

みなさんも、sample03 のコードを見ながら run メソッドのアクティビティ図を描いてみましょう。同じような図が描けたら、いったん保存して、次の説明へ進みましょう。

【コラム】アクティビティと実装を対応づける

もし、図 14 の最初のアクティビティを「ライン上か外れているか調べる」と書いてあつたとしましょう。そのとき、実装がリスト 5 であつたら、どんなことが問題でしょうか。

この場合、実装した人が「センサーの値としきい値の比較すればいいんだな」と実現方法を頭に浮かべて実装したことになります。つまり、「ライン上か外れているか調べる」ための方法が「センサーの値としきい値の比較」だと、実装した人が頭の中で紐づけたのです。その結果、モデル図とコードのどちらにも、頭の中で作ったつながりが残らなくなってしまったのです(これこそ設計情報であるのに!)。

もちろん、手順を踏んで設計するならば、最初のアクティビティは「ライン上か外れているか調べる」と書く方が望ましいでしょう。しかし、そのときの実装は、リスト 5 ではまだ十分とはいえないのです。

では、どのようなモデル図と実装が望ましいでしょう。考えてみてください。

走りの改善策を検討する(sample04)

sample03 の走行を見直すために、実験してみましょう。

sample03の走り方の課題はなにか

sample03 のライントレースをよくみると、次のような課題が見つかります。

- ・ ライン上かライン外かで左右のモーターのどちらかを止めている
- ・ モーターを止めた車輪を軸にして旋回するため、大きく左右に揺れる
- ・ 速度を上げると揺れが大きくなり、走行が不安定になってしまう

どうやら、もう少し滑らかに走る方法を考えてみた方がよさそうだとわかつてきました。

滑らかに走る方法を考えてみよう

もう少し滑らかに走れるようにするには、どんなことを考えればよいでしょうか。

sample03 の走り方は、ライン上かライン外かで動作が2種類だけです。ラインからたくさん外れたときも、少ししか外れていないときも、同じ動作になります。走り方を滑らかにするには、ラインから外れた度合いに応じて処理を変化させる必要がありそうです。

そこで、ラインの境界付近でのカラーセンサーの値の変化を考えてみます。

- ・ カラーセンサーの取得値は、白と黒の間の値で徐々に変化しているのではないか
- ・ ラインの境界付近では、ライン上の値寄りの値になっていないか
- ・ 白と黒の間の値が取得できれば、その分旋回する度合を小さくできないか

どうやら、ラインの境界付近の動作を調整して旋回する度合いを小さくできれば、もっと滑らかにできそうです。

- ・ 測定値と比較する目標値を、白と黒の間の値にしておき、測定値と目標値の差をラインからずれた度合いとみなす
- ・ 動作を切り替えるとき、PWM値を0とせず、測定値の大きさに応じて減らしてみる

どうやら、カラーセンサーの測定値と比較する目標値の差の大小に応じて左右のモーターのPWM値を加減すれば、ズレに応じて旋回の度合いを調整できそうです。

プログラムで使えるよう整理しよう

いま考えているような制御の方法を「比例制御(Proportional制御, P制御)」と呼びます。これまで考えたことを計算できるよう、式に整理してみましょう。

- ・ 偏差 = カラーセンサーの測定値 - カラーセンサーの目標値
- ・ 調整値 = 偏差に比例した値(偏差と調整用の係数をかけた値)
- ・ 調整後の左のモーターのPWM値 = 基準のPWM値 - 調整値
- ・ 調整後の右のモーターのPWM値 = 基準のPWM値 + 調整値

比例制御では、調整値を求めるときに使う係数を「比例ゲイン(K_p)」、調整後の値(このサンプルでは調整後の左右のモーターのPWM値)を「操作量」といいます。

うまくいきそうか確かめてみよう

アイディアは整理できましたが、まだうまくいか確かめられていません。

- ・この方法には効果があるのか
- ・どのようにプログラムを作れば実現できるのか
- ・目標値や係数をどのくらいの値にすればよいのか

アイディアの妥当性を確かめるには、やはり「実験して確かめる」のがよいでしょう。

比例制御を実験する

考えたことを実際に試してみて、効果や実現方法を決めましょう。

1. サンプルコードの `sample03` ディレクトリをそっくりコピーして `sample04` ディレクトリを作る
2. 考えた方法に合わせて `app/Tracer.cpp` の `run` メソッドを編集する
3. 編集が済んだら、期待したとおりに動くか実験してみる
4. うまくいかない時は、定数の値を変更して実験を繰り返す

作成したコードは [リスト 12](#) のようになりました。定数は、みなさんが調整してみてください。

リスト 12. sample04/app/Tracer.cpp (runメソッドの部分を抜粋)

```
18 void Tracer::run() {  
19     const float Kp = 0.83; ①  
20     const int target = 10; ②  
21     const int bias = 0;    ③  
22  
23     msg_f("running...", 1);  
24     int diff = colorSensor.getBrightness() - target; ④  
25     float turn = Kp * diff + bias; ⑤  
26     leftWheel.setPWM(pwm - turn); ⑥  
27     rightWheel.setPWM(pwm + turn); ⑥  
28 }
```

- ① 比例係数(1より小さい値で検討しましょう)
- ② 白と黒の中間値を目標値とする(測った値があれば、それを使いましょう)
- ③ 調整値にいつも一定の値(バイアス)を追加しておきたい場合に設定する
- ④ 偏差を求める
- ⑤ 調整値を求める
- ⑥ 左右の車輪に調整後の操作量を与えた

編集できたら、動かしてみましょう。

みなさんも実験してみましょう

みなさんも **sample04** を作って実験してみましょう。うまく動作するようプログラムを調整できたら、いったん保存して、次の説明へ進みましょう。

まだ実験しか終わっていなかった!

さて、実験もうまくいったことだし、これでオッケーでしょうか。

次のことはどう考えればよいでしょうか。

- ・ sample04 の実験で実現したことは、モデル図のどこかに描いてありますか
- ・ sample04 の結果は、sample03 のモデル図のどこを直せば表現できるでしょうか

実験は成功しましたが、その結果をシステムに組込んだとは、どうやらまだいえなさそうです。

実験成果をモデルとコードに反映する(sample05)

sample03 のモデル図を元に、sample04 で実験した結果を反映した sample05 のモデル図を作成しましょう。

実験成果を反映したモデル図を作成する

1. sample03 のモデル図を複製して sample05 を作る
2. クラス図の名前を sample05 のクラス図 に変更する
3. アクティビティ図を編集して、実験で確立した処理を反映する
 - 比例制御の調整値の計算をしているところ
 - 実際に走行させるところ
4. アクティビティ図の変更に対応して、調整値の計算を run メソッドから分離する
 - 調整値の計算を分けて、calc_prop_value メソッドとして追加する

実験成果を反映したアクティビティ図

Tracer クラスの `run` メソッドのアクティビティ図は、図 15 のようになるでしょう。

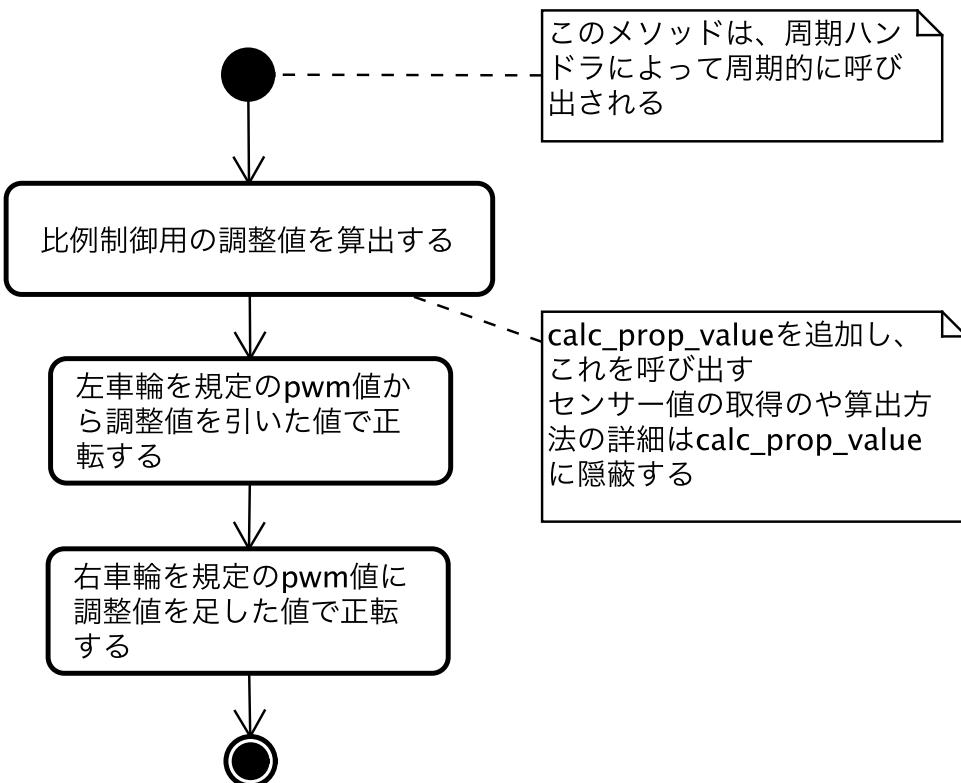


図 15. samaple05の Tracer クラスの `run` メソッドのアクティビティ図

実験成果を反映したクラス図

Tracer クラスのクラス図は、図 16 のようになるでしょう。Tracer クラスに private な操作 calc_prop_value を追加しました。

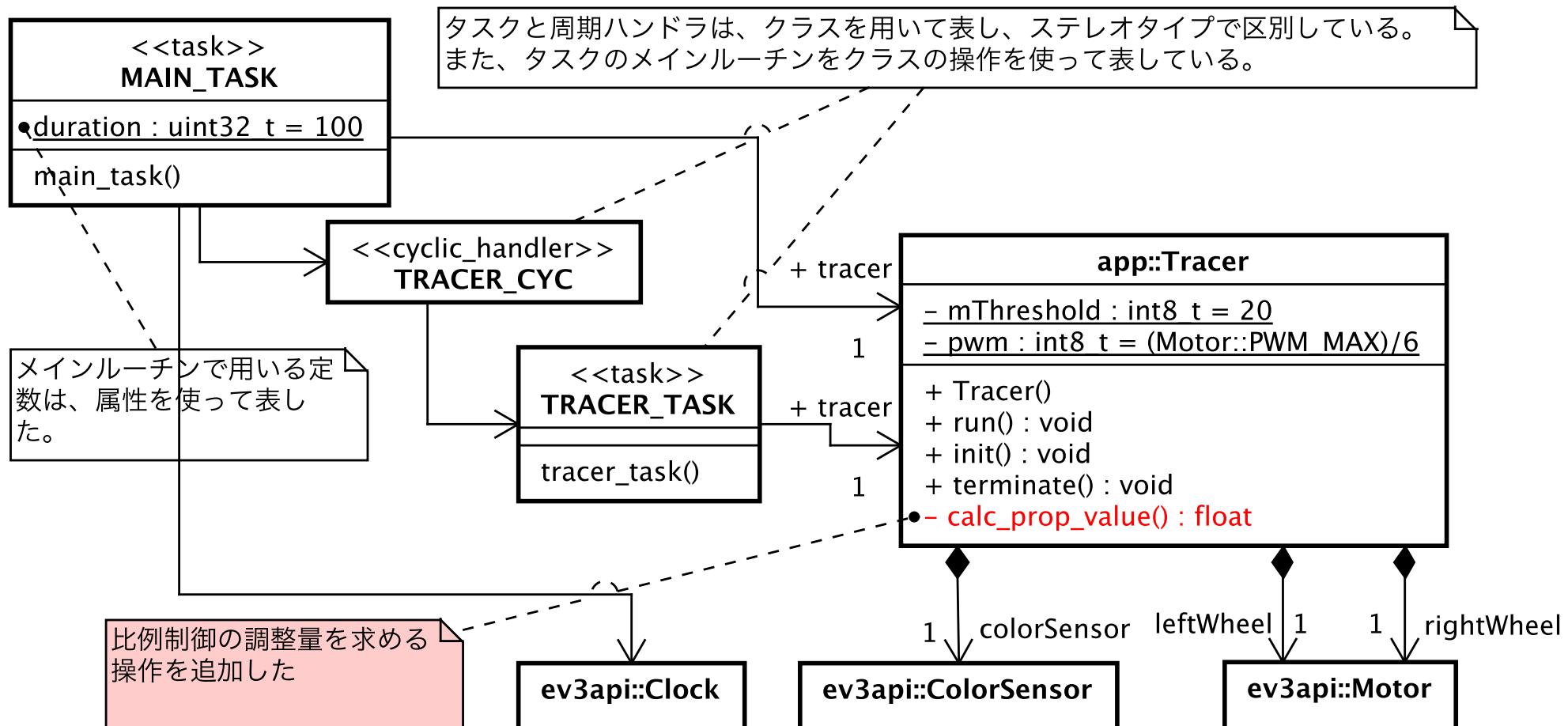


図 16. sample05 のクラス図

みなさんもモデル図を更新しましょう

みなさんも、sample05 のモデル図を更新しましょう。同じような図が描けたら、いったん保存して、次の説明へ進みましょう。

モデル図に対応したコードを作成する

更新したモデル図に対応したプログラムを作成しましょう。

1. サンプルコードの `sample04` ディレクトリをそっくりコピーして `sample05` ディレクトリを作る
2. クラス図に従って `Tracer.h` に `calc_prop_value` メソッドを追加する
3. `Tracer.cpp` にも `calc_prop_value` メソッドを追加する
4. `Tracer` クラスの `run` メソッドを `calc_prop_value` メソッドを使うように修正する

作成したコードは リスト 13 ~ リスト 15 のようになりました。

リスト 13. sample05/app/Tracer.h

```
14 private:  
15     Motor leftWheel;  
16     Motor rightWheel;  
17     ColorSensor colorSensor;  
18     const int8_t mThreshold = 20;  
19     const int8_t pwm = (Motor::PWM_MAX) / 6;  
20  
21     float calc_prop_value();    ①  
22 };
```

- ① calc_prop_value メソッドを追加した

リスト 14. sample05/app/Tracer.cpp (calc_prop_value を抜粋)

```
18 float Tracer::calc_prop_value() {  
19     const float Kp = 0.83;           ①  
20     const int target = 10;          ②  
21     const int bias = 0;  
22  
23     int diff = colorSensor.getBrightness() - target; ③  
24     return (Kp * diff + bias);      ④  
25 }
```

- ① sample04 の実験結果を反映する
- ② sample04 の実験結果を反映する
- ③ 偏差求める
- ④ 調整値を求める

リスト 15. sample05/app/Tracer.cpp (run メソッドを抜粋)

```
27 void Tracer::run() {  
28     msg_f("running...", 1);  
29     float turn = calc_prop_value(); ①  
30     int pwm_l = pwm - turn;        ②  
31     int pwm_r = pwm + turn;        ②  
32     leftWheel.setPWM(pwm_l);  
33     rightWheel.setPWM(pwm_r);  
34 }
```

- ① 比例制御の調整値を求める
- ② 基準値と調整値を使って操作量を求める

編集できたら、動かしてみましょう。

みなさんも実験してみましょう

みなさんも sample05 を作って実験してみましょう。うまく動作するようプログラムを調整できたら、いったん保存して、次の説明へ進みましょう。

ここまでまとめ

- 初歩的なライントレーサーを動かした([sample03](#))
 - ファイルをクラスごとに分けてあった
 - ディレクトリを使って構造を分けてあった
 - 周期ハンドラを使って周期的に処理していた
- [sample03](#) をモデル図で表した
 - パッケージにディレクトリを割当てた
 - タスクと周期ハンドラをクラスに見立てた
 - ライントレーサーの振舞いをアクティビティ図で表した
- 滑らかに走行するためのアイディアを実験した([sample04](#))
 - カラーセンサーの値の変化について検討した
 - 調べた結果を使って滑らかに走行できるか実験した
- 実験した成果をモデルとコードに反映した([sample05](#))
 - 実験成果をモデル図に反映した
 - モデル図に対応するコードを作成し、動作を確認した

要素技術をモデル図に表すには

組込みシステムが安定して精度良く動作するためには、制御やセンシングに要素技術を活用することが欠かせません。そして、モデル図を使えば、どのような要素技術をシステムのどこにどのように組み込んだのか示せます。

この章の演習を通じて、そのことが少し実感できたのではないかでしょうか。

要素技術を取り入れるときのポイント

要素技術を取り入れるときに注意すべきポイントを整理しておきましょう。

- 要素技術を確保できただけで終わりにしない
 - 実験してうまくいっても、そのままにせず、設計に戻ること
- 要素技術をモデル図の構成要素にする
 - 要素技術をモデルの構成要素(クラスや操作)に割り当てる
 - 要素技術の動作(処理の流れなど)を振舞いのモデルで表すこと
- 構成要素の名前がモデル図の上に見える
 - 要素技術を使っていることがわかるクラス名や操作名をつける
 - 名前をつけて呼び出すことができていて、初めて使われているとわかる

要素技術をシステムに組込む手順

次のような検討手順で進めてみるとよいでしょう:

1. アイディアを出す、必要な情報を取得する・調査する
2. 実験してアイディアの効果を確かめる
3. 得られた結果を、元のシステムのモデル図に「名前をつけて」組込みむ
4. 更新したモデル図に合わせてコードを作成する

本資料について

資料名：要素技術とモデルを開発に使おう：要素技術をシステムに組込もう（技術教育資料）

作成者：© 2016,2017,2018,2019,2020 by ETロボコン実行委員会

この文書は、技術教育「要素技術とモデルを開発に使おう」に使用するETロボコン公式トレーニングのテキストです。

3.0, 2020-06-29 11:13:54, 2020年用