- モデルからソフトウェアを作るためには、ソースコードへの変換が必要です
- 3章で作成したモデルを、どのようにしてソースコードに変換するのか、詳しくみてみましょう

4. モデルからソースコードを作る

4-1. プログラムとソースコード

- プログラム
  - システム上で実際に動くソフトウェア
    - 対象システムだけでなく同時に使用するOSやライブラリを合わせて指すこともあります
- プログラミング言語
  - アイディアや設計書をプログラムに変換するために使う表記法
  - この演習では「C++」を使用します
    - 組込みソフトウェア開発でデバイスを直接動かすときは「C言語」や「C++」がよく使われています
    - EV3では、JavaやC#用の機器制御用のライブラリが提供されているので、これらを使って作ることもできます
- ソースコード
  - プログラムの機能・構造・振舞いをプログラミング言語で表現したもの
- 実装
  - アイディアや設計書からソースコードに変換する作業や、変換して得たソースコードを指します
    - プログラミング言語やプログラムを実行する環境によって、ソースコードに書くことは異なります
    - 人間が手で変換する方法（プログラミング）とツールによる自動変換があります

4-2. 実装モデルに変換する（１）

- これまでのモデル
  - 利用者、設計者が理解できることばで書いていました
    - モデルの中では日本語（対象システムのことば）を使っていました
  - 実装に使う環境や使える資源を反映できていません
    - 環境や実装方法を意識しないで作っていました

- 実装するときに使えるモデルが必要です
  - プログラミング言語で扱える表現が必要です
  - 用意されている資源とのつながりを表すことが必要です
    - OSのシステムコールや、ライブラリのAPIなど
  - 実装上必要になる条件や技術を使う必要があります
    - プログラム上にデータを保持する構造や方式の選択
    - OSに依存するタスクのコンフィギュレーションや、ハンドラ等の設定

4-2. 実装モデルに変換する（2）

EV3

詳細設計

- 構造に関するアーキテクチャを整理します
  - app（ライントレーサ）、unit（ライン監視部や倒立走行部）、platform（OSやEV3RT C++など）に分け、それぞれをパッケージとして扱いましょう
- 提供されたライブラリのクラスやAPIに読み替えます
  - センサ類やモータの操作には、EV3はEV3RT C++ APIを使うことにします
- C言語用に提供されているAPI群を、C++のクラスとメソッドに再構成します
  - 3章で考えたことを反映できるよう、C言語用に提供されている倒立振子制御ライブラリをそのまま使わずに、C++のクラスに仕立てて使いましょう
- 自分たちが案出したクラスをプログラミング言語で使える表現に直します
  - ライントレーサ：LineTracer、ライン監視部：LineMonitor、倒立走行部：BalancingWalker、などとします
    - 今回のように設計モデルと実装モデルが1対1にならず、実装モデルの複数のクラスに対応づけられる場合もあります

この演習の構造に関するアーキテクチャを図として整理したものは付録a-5.「演習に使った構造のアーキテクチャ」にまとめてあります。
以降のモデル図に使っている色は、この付録の図の領域の色に対応してあります。

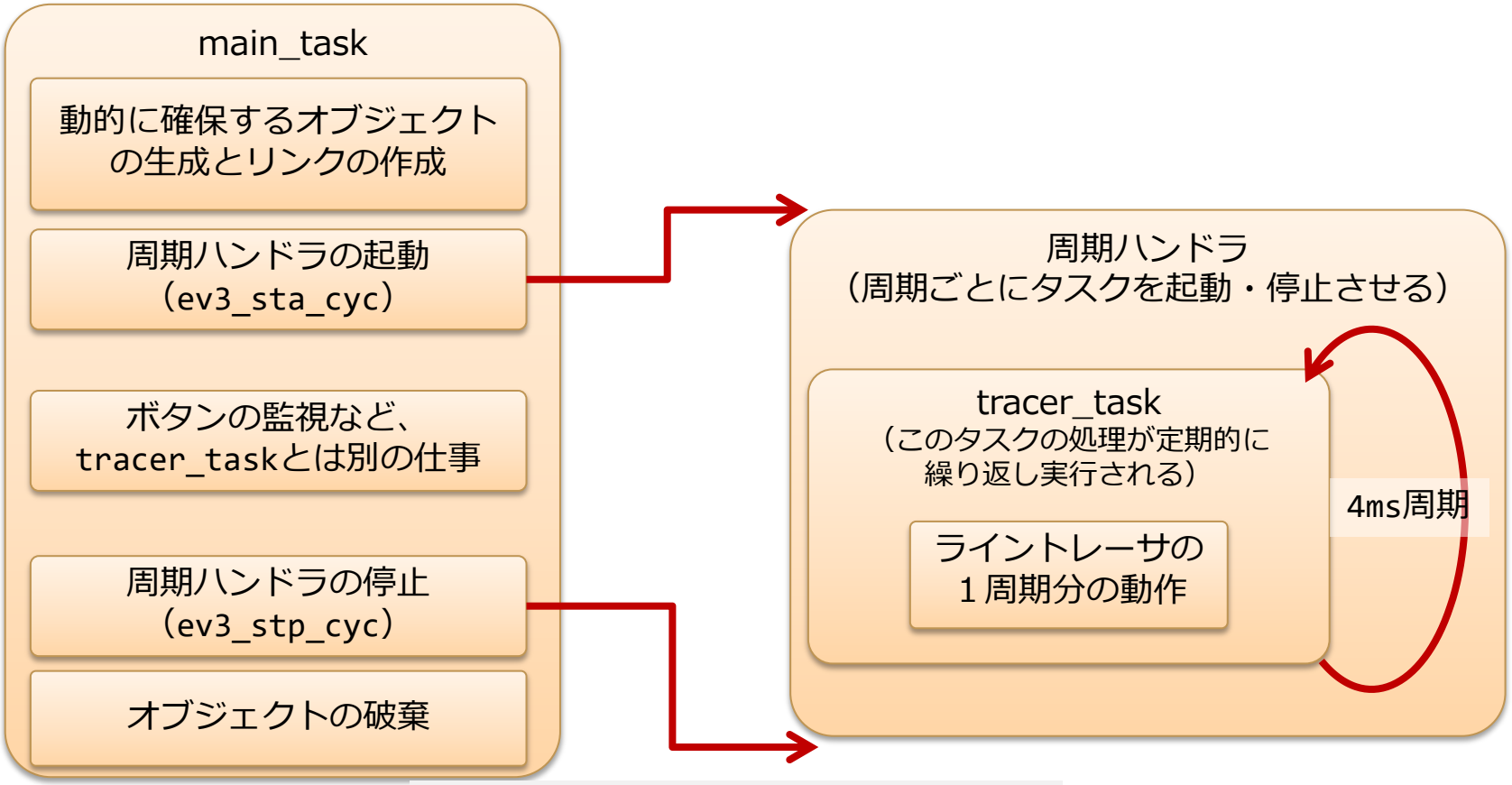
この演習で使用する命名規則は付録a-3.「サンプルコード命名規則」にまとめてあります

4-2. 実装モデルに変換する（3）

EV3

詳細設計

- タスクを周期的に動かす方式で実装することに決めましょう
  - EV3RTでは、タスクを活性化しておき、OSの周期ハンドラの起動・停止を使います
- これで、4ms周期で動作させるしくみをどうやって実現するかが決まりました



タスクと周期ハンドラの構成（EV3RTの場合）

4-3. 実装モデルを作成する（1）

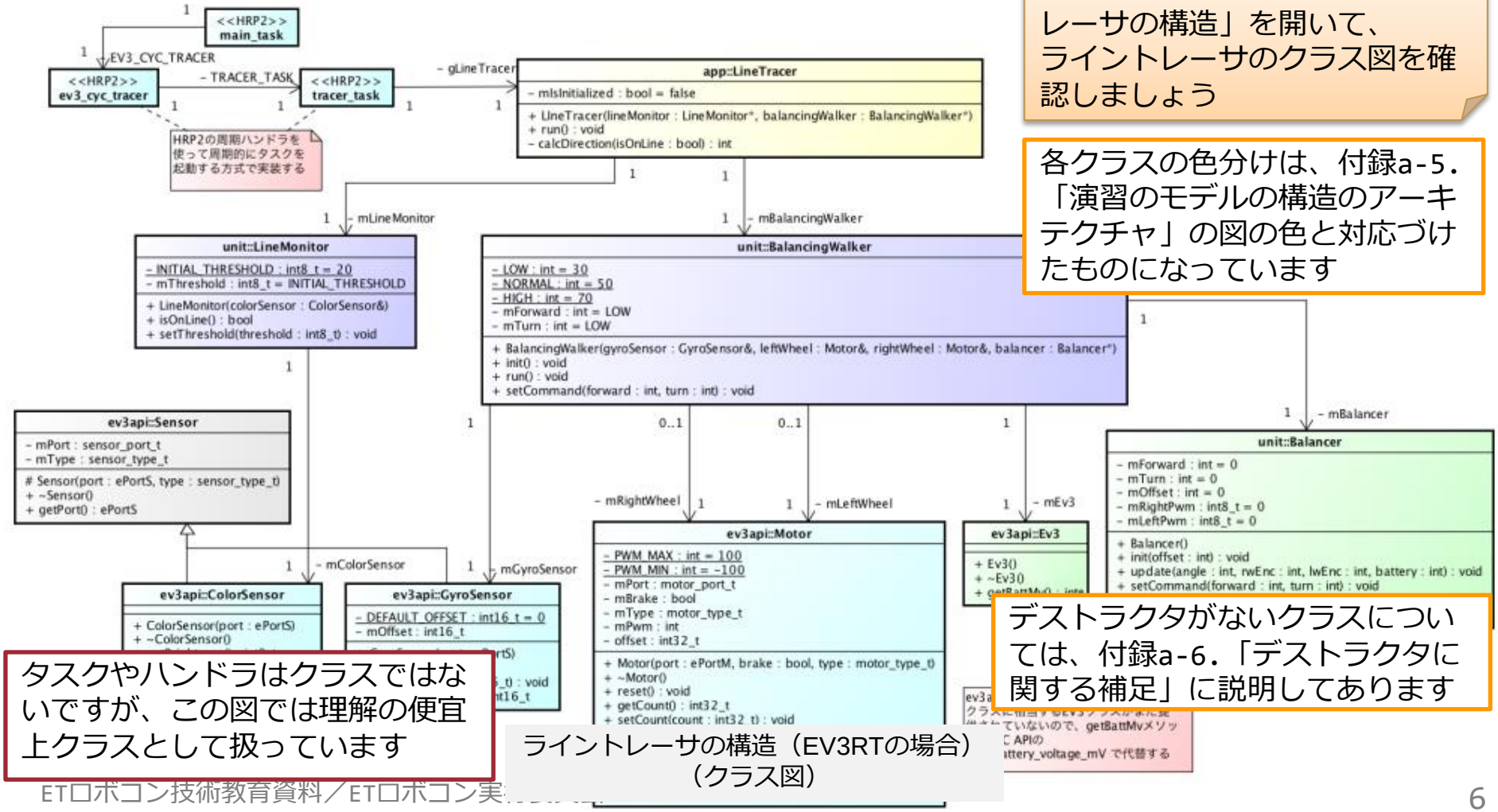
EV3 ET ROBOT CONTEST

詳細設計

- 変換の結果をまとめて実装モデルのクラス図を作成します
  - 設計モデルの構造＋構造のアーキテクチャ＋実装モデルへの変換の結果を組合わせると、実装用の構造のモデルができます

実装モデル・構造「ライントレーサの構造」を開いて、ライントレーサのクラス図を確認しましょう

各クラスの色分けは、付録a-5.「演習のモデルの構造のアーキテクチャ」の図の色と対応づけたものになっています



デストラクタがないクラスについては、付録a-6.「デストラクタに関する補足」に説明してあります

タスクやハンドラはクラスではないですが、この図では理解の便宜上クラスとして扱っています

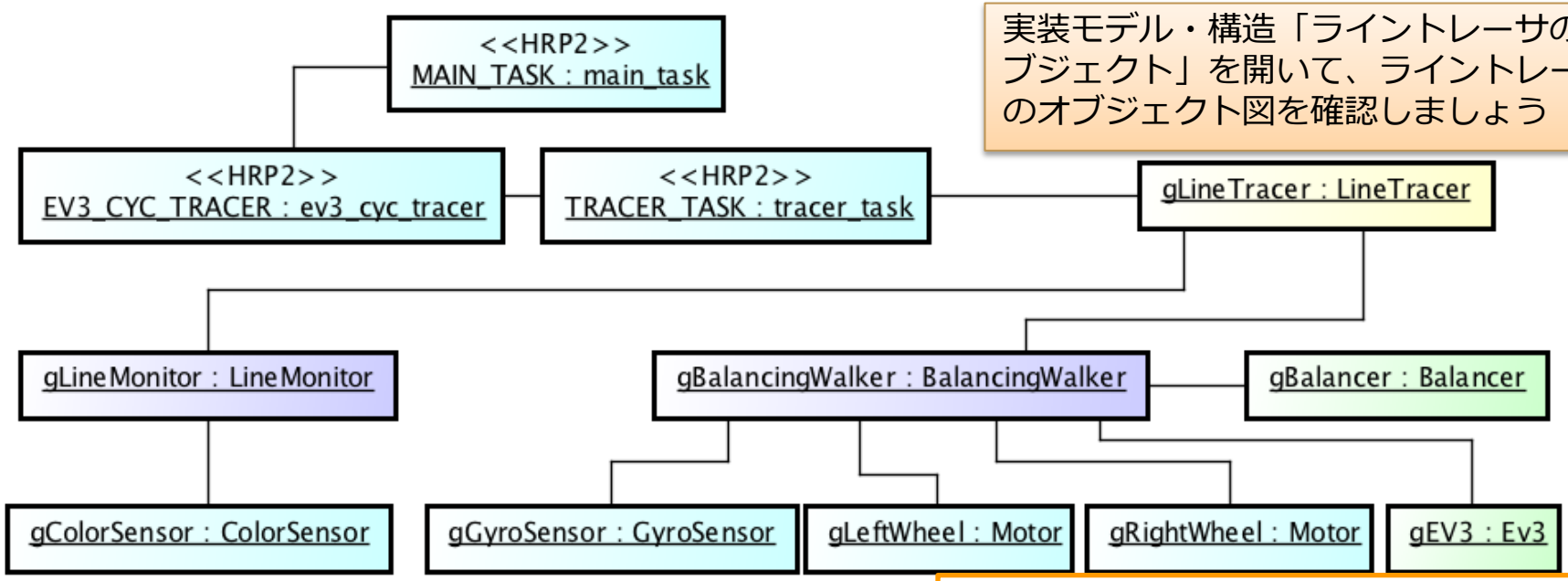
ライントレーサの構造（EV3RTの場合）（クラス図）

4-3. 実装モデルを作成する（2）

詳細設計

- 実装モデルのオブジェクト図を作成します
  - 設計モデルのオブジェクト図に登場することばを、実装モデルのクラス図で用意したことばで置き換えます
  - オブジェクト名は、実装するときにオブジェクトを作成するときに使う名前になります

実装モデル・構造「ライントレーサのオブジェクト」を開いて、ライントレーサのオブジェクト図を確認しましょう



タスクやハンドラはクラスではないので、クラスのインスタンスとして扱うのは少しおかしいですが、この図では理解のためにそのように扱っています

各オブジェクトの色分けは、付録a-5.「演習のモデルの構造のアーキテクチャ」の図の色と対応づけたものになっています

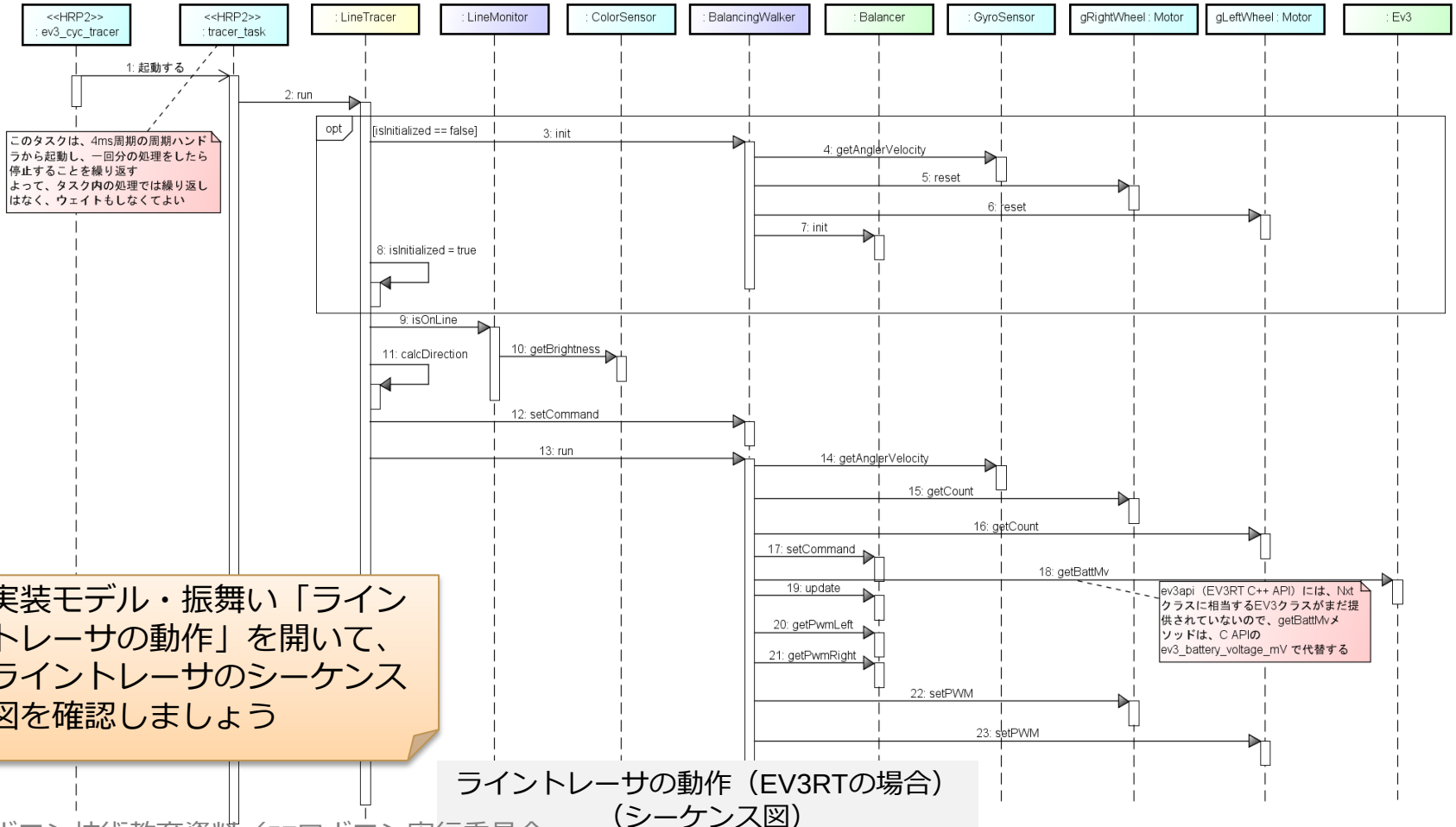
ライントレーサの構造（EV3RTの場合）
（オブジェクト図）

4-3. 実装モデルを作成する（3）

EV3 ET ROBOT CONTEST

詳細設計

- 実装モデルのシーケンス図を作成します
  - 設計モデルのシーケンス図に登場することばを、実装モデルのクラス図で用意したことばで置き換えます



実装モデル・振舞い「ライントレーサの動作」を開いて、ライントレーサのシーケンス図を確認しましょう

ライントレーサの動作（EV3RTの場合）
（シーケンス図）

4-4. ソースコードの構成を考える（1）

詳細設計

- 実装モデルを使ってソースコードを作成します
  - UMLのモデル図とソースコードの関係は、次のようになります

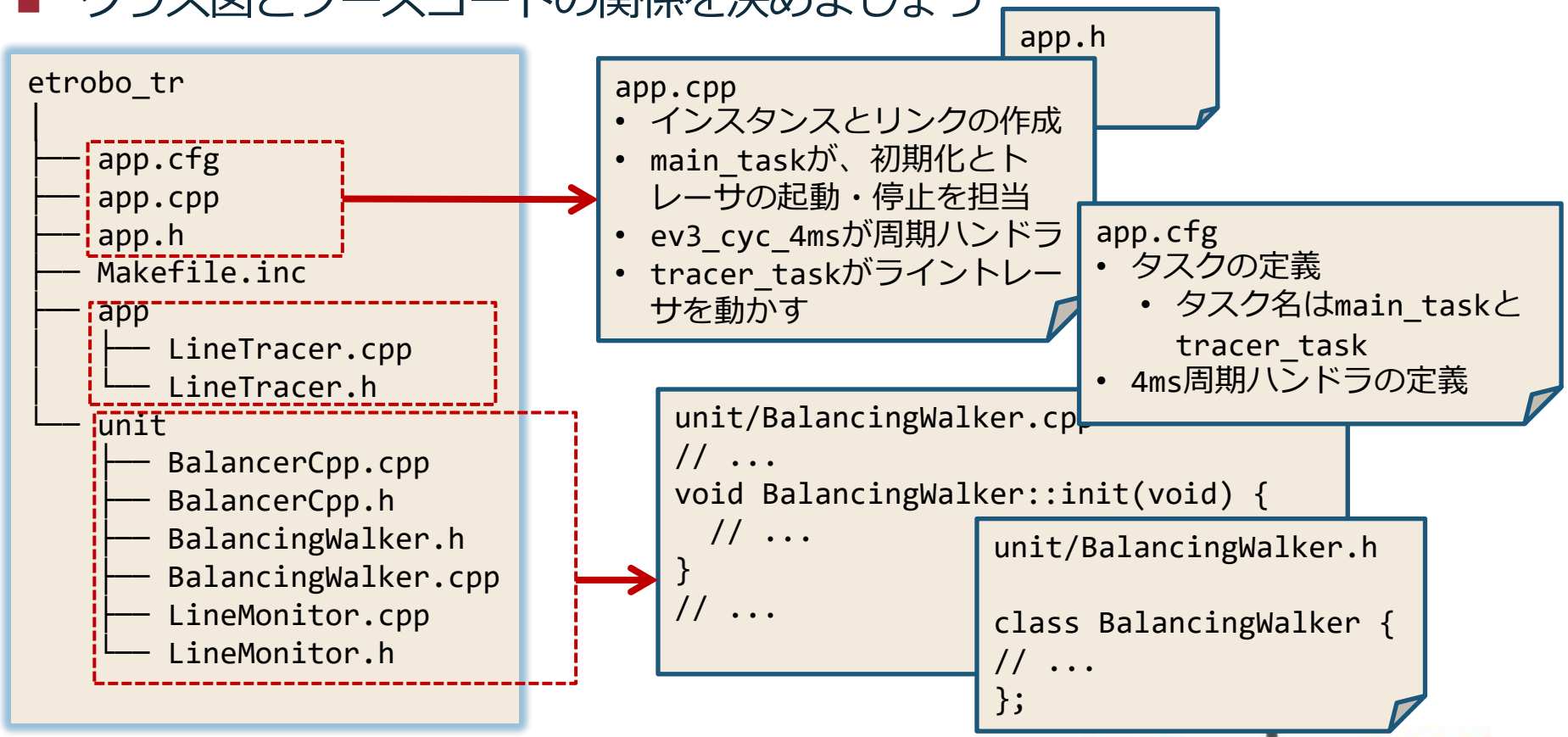
順序	視点	よく使うモデル図	モデルの役割	実装するコード
1	システムの機能	ユースケース図	モデリングの対象となるシステムが提供する機能を記述します	直接コードに変換しません（部品の動きの組合わせで実現します）
2	システムの振舞い	アクティビティ図 ステートマシン図	機能を実現するための制御フローや、状態に応じたシステムの動作を記述します	
3	システムが動くときの構造	オブジェクト図	機能を実現するために必要な部品を定義します	各クラスのインスタンスを作成するファイルを作ります
4	システムの構築に必要な構造	クラス図	オブジェクト図で記載された各部品の仕様を定義します	*.hファイルにクラスを定義します
5	システムの内部の要素間の振舞い	コミュニケーション図 シーケンス図	機能を実現するための、部品どうしの協調動作を記述します	*.hファイルに他のクラスとの関連を追加します
6	システムの内部のある要素の振舞い	アクティビティ図 ステートマシン図	部品の操作に対する制御フローや、状態に応じた部品内部の動作を記述します	*.cppファイルにメソッドを定義します

4-4. ソースコードの構成を考える（2）

EV3

詳細設計

- 実装モデルのパッケージに合わせて、コードを作成するディレクトリを決めましょう
- 実行時のインスタンス構築やタスク構成を整理しましょう
- クラス図とソースコードの関係を決めましょう



4-5. クラスを実装するときの原則

実装

- モデル図を見ながらソースコードを作成します
  - オブジェクト（クラスのインスタンス）はクラスを使って作成しますので、先にクラスを作成します
  - クラス図に対応するようにクラスを作成します
  - 既存のクラスは参照し、新たに作成しないことに注意しましょう
- クラス図からソースコードへの変換手順を決めて、どのクラスも同じように対応づけましょう
  - 対応づけによって、クラス図とクラスの実装の対応関係が維持できるようになります
    - 実装時に設計を見直すべきとき、クラス図に戻って検討できるようになります
- 特別扱いのクラス
  - 別の変換方法で作成するクラスがあるときは、そのクラスだけ特別扱いしてください
    - 特別扱いが必要なクラスには、実装クラス図上にノートやステレオタイプを使って、特別扱いすることがわかるようにしておきます

演習で使った命名規則は付録a-3.「サンプルコード命名規則」にまとめてあります

4-6. ソースコードを書く（1）

実装

- ひとつのクラスを1組のソースコードに対応づけます
  - 実装モデルのクラス名をもとにヘッダファイル「クラス名.h」と実装ファイル「クラス名.cpp」を作成します
  - ヘッダファイルには「インクルードガード」を書いておきます
- ヘッダファイルに、属性と操作の宣言を追加します
  - 属性はメンバ変数として、操作はメソッドとして型を合わせて追加します
- 関連するクラスのヘッダファイルをヘッダファイルにインクルードします
  - 直接関連するクラスと内部で使用するライブラリのヘッダだけをインクルードします
- クラスの属性に、関連するクラスのインスタンスへの参照を追加します
  - 関連につけた関連端名を属性に使います
- 実装ファイル（cppファイル）に、ヘッダファイルをインクルードします
  - 実装するクラス自身のヘッダファイルをインクルードします
    - このクラスを利用するクラスが必要とするヘッダファイルは、 cppファイルではなく、このクラスのヘッダファイルにインクルードしておきます
- クラスの操作を参照して、cppファイルに対応するメソッドを追加します

インクルードガードと、今日的な対応方法については、付録で説明しています

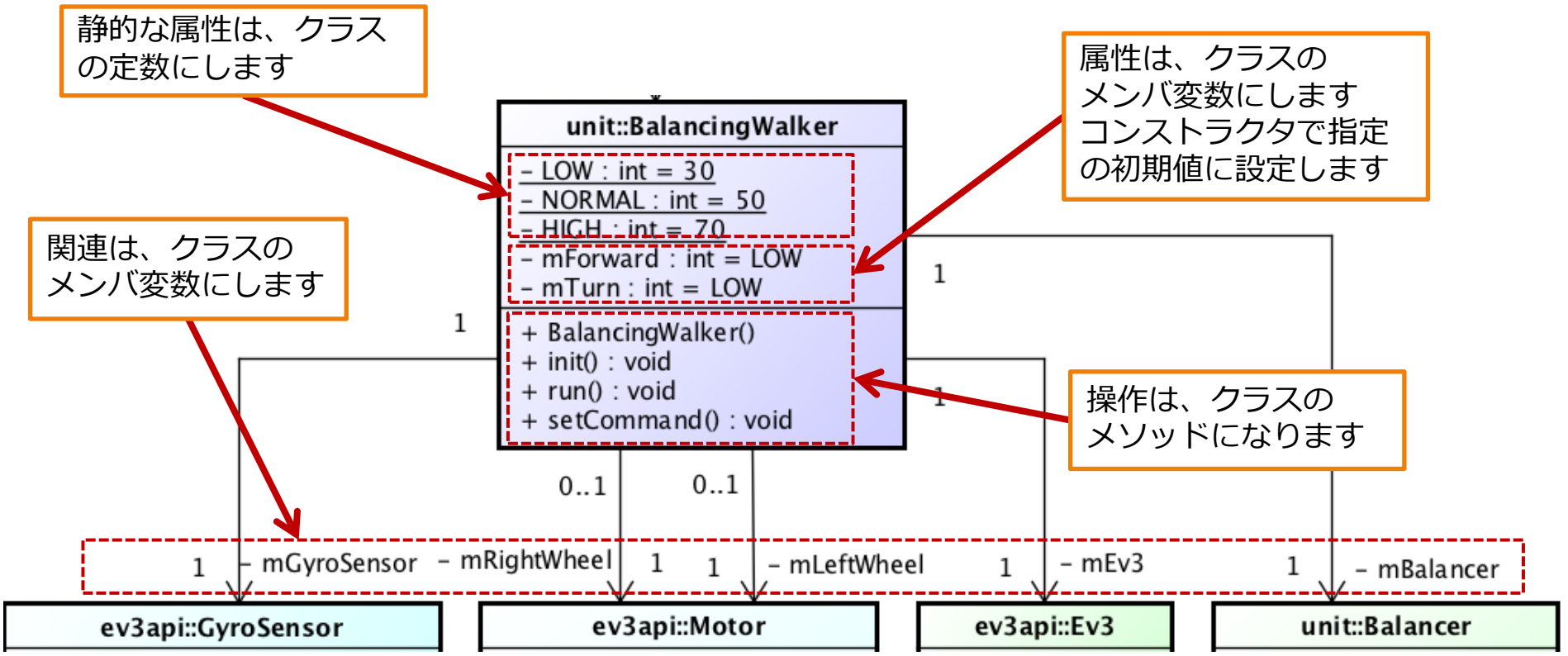
4-6. ソースコードを書く（2）

EV3 ET ROBOT CONTEST

実装

- クラスとソースコードの関係を確認しましょう
  - BalancingWalkerの例

実装モデル・構造「ライントレーサの構造」を開いて、クラスの詳細や関連端を確認しましょう



構造とソースコードの関係（EV3RTの場合）

4-6. ソースコードを書く（3）

EV3 ET ROBOT CONTEST

実装

- ヘッダファイルを作成しましょう
  - BalancingWalkerの例

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう

unit/BalancingWalker.h

```
#ifndef EV3_UNIT_BALANCINGWALKER_H_
#define EV3_UNIT_BALANCINGWALKER_H_

#include "GyroSensor.h"
#include "Motor.h"
#include "BalancerCpp.h"

class BalancingWalker {
public:
  static const int LOW;
  static const int NORMAL;
  static const int HIGH;

  BalancingWalker(
    const ev3api::GyroSensor& gyroSensor,
    ev3api::Motor& leftWheel,
    ev3api::Motor& rightWheel,
    Balancer* balancer);

  void init();
  void run();
  void setCommand(int forward, int turn);

private:
  const ev3api::GyroSensor& mGyroSensor;
  ev3api::Motor& mLeftWheel;
  ev3api::Motor& mRightWheel;
  Balancer* mBalancer;
  int mForward;
  int mTurn;
};

#endif  // EV3_UNIT_BALANCINGWALKER_H_
```

クラスの操作

他のクラスへの関連

クラスの定数

クラスの属性

インクルードガードと、今日的な対応方法については、付録で説明しています

4-6. ソースコードを書く（4）

EV3 ET ROBOT CONTEST

実装

- CPPファイルを作成しましょう
  - BalancingWalkerの例

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう

unit/BalancingWalker.cpp

```
#include "BalancingWalker.h"

const int BalancingWalker::LOW    = 30;
const int BalancingWalker::NORMAL = 50;
const int BalancingWalker::HIGH   = 70;

BalancingWalker::BalancingWalker(
  const ev3api::GyroSensor& gyroSensor,
  ev3api::Motor& leftWheel,
  ev3api::Motor& rightWheel,
  Balancer* balancer)
  : mGyroSensor(gyroSensor),
    mLeftWheel(leftWheel),
    mRightWheel(rightWheel),
    mBalancer(balancer),
    mForward(LOW),
    mTurn(LOW) {
}

// 続く
```

クラスの定数に初期値を設定します

関連するクラスのインスタンスとのリンクは、コンストラクタの初期化リストで設定します

メンバ変数は、コンストラクタの初期化リストで指定の初期値に設定します

初期化リストを使ったコンストラクタの書式については、付録で説明しています

4-6. ソースコードを書く（5）

EV3

実装

- CPPファイルを作成しましょう
  - BalancingWalkerの例

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう

unit/BalancingWalker.cpp

```
// 続き

void BalancingWalker::run() {
  int16_t angle
    = mGyroSensor.getAnglerVelocity();
  int rightWheelEnc
    = mRightWheel.getCount();
  int leftWheelEnc
    = mLeftWheel.getCount();

  mBalancer->setCommand(mForward, mTurn);
  int battery = ev3_battery_voltage_mV();
  mBalancer->update(
    angle, rightWheelEnc,
    leftWheelEnc, battery);

  mLeftWheel.setPWM(
    mBalancer->getPwmLeft());
  mRightWheel.setPWM(
    mBalancer->getPwmRight());
}
```

```
void BalancingWalker::init() {
  int offset
    = mGyroSensor.getAnglerVelocity();

  mLeftWheel.reset();
  mRightWheel.reset();
  mBalancer->init(offset);
}

void BalancingWalker::setCommand(
  int forward, int turn) {
    mForward = forward;
    mTurn    = turn;
}
```

4-7. オブジェクト図を実装する（1）

EV3 ET ROBOT CONTEST

実装

- オブジェクト図で使用しているオブジェクトを作成します
  - app.hに、アプリケーションのクラスのヘッダファイルをインクルードします
  - app.cppで、センサとモータのオブジェクトを作成します
    - センサとモータのライブラリは、使用上の制約があります
      - この演習では、オブジェクトは静的に（newしないで）グローバルに作成する方法を使います
        （EV3RT beta3-1までのセンサやモータのオブジェクトは動的に（newして）作成する制約がなくなりました）
    - EV3RT C++ APIライブラリが提供するコンストラクタを使います
  - unitやappに含まれるライントレーサ固有のクラスのオブジェクトを作成します
- オブジェクト間のリンクを作成します
  - リンクは関連のインスタンスです
    - クラス間に関連が引かれているとき、オブジェクト間に引くものがリンクです
    - 他のクラスへの参照を保持する属性が、関連先クラスのオブジェクトへのリンクを保持します
  - あらかじめ関連先のクラスのオブジェクトを作成しておきます
  - コンストラクタの初期化リストで初期化します
    - メンバ変数を初期化するときに、関連先のオブジェクトを使って初期化することでリンクが作成されます
    - 初期化時以外でも、関連先の参照を保持する属性を書き換えればリンクの作成や変更ができます

4-7. オブジェクト図を実装する（2）

EV3 ET ROBOT CONTEST

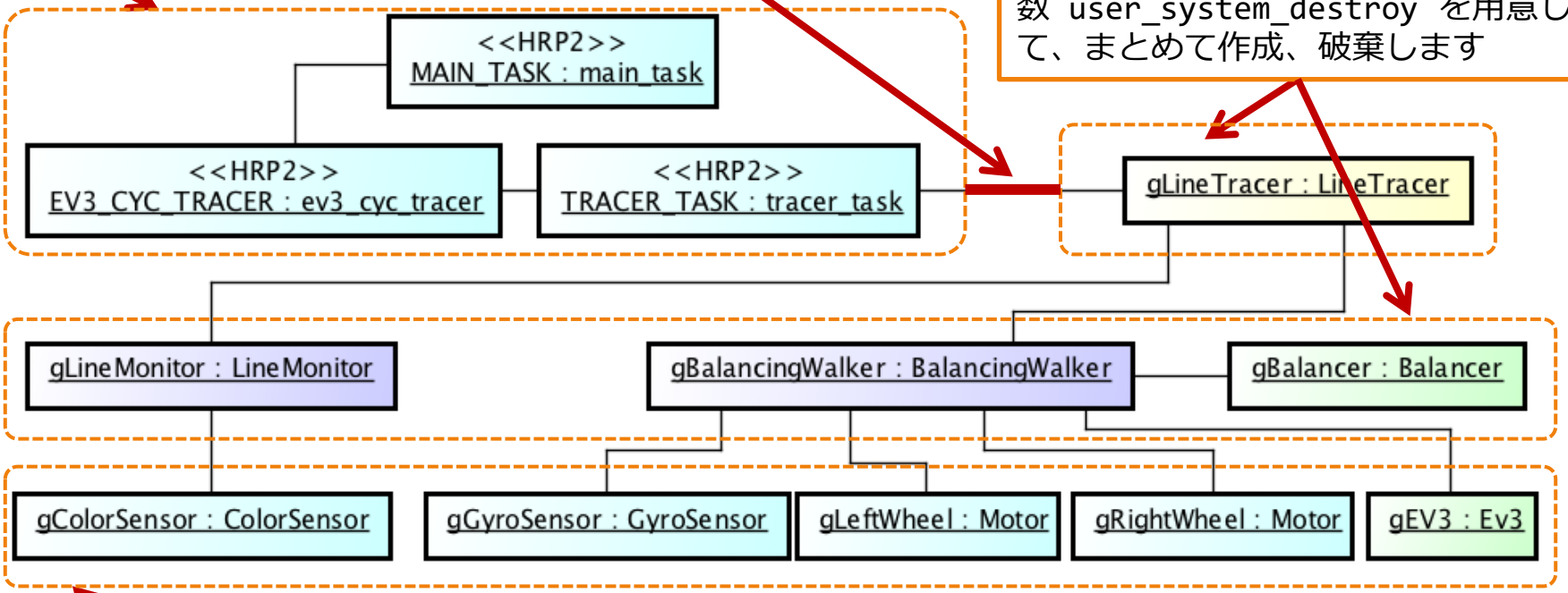
実装

- オブジェクト図とソースコードの関係を確認しましょう

タスクの作成と起動、インスタンスの作成・破棄は、app.cpp が担当します

app.cが依存するのはここ
```
#include "LineTracer.h"
```

appとunitにはクラスのインスタンスとリンクを動的に作成する関数 user_system_create と破棄する関数 user_system_destroy を用意して、まとめて作成、破棄します



デバイスのクラスのインスタンスは静的に作成します
（nxtOSEKのC++ライブラリを使う時は制約があります）
ex) LightSensor gLightSensor(PORT_3);

ライントレーサのオブジェクト図（nxtOSEKの場合）

4-7. オブジェクト図を実装する（3）

実装

- オブジェクト図とソースコードの関係を確認しましょう

app.cpp

```
#include "app.h"
#include "LineTracer.h"

#if defined(BUILD_MODULE)
#include "module_cfg.h"
#else
#include "kernel_cfg.h"
#endif

using ev3api::ColorSensor;
using ev3api::GyroSensor;
using ev3api::Motor;

ColorSensor gColorSensor(PORT_3);
GyroSensor  gGyroSensor(PORT_4);
Motor       gLeftWheel(PORT_C);
Motor       gRightWheel(PORT_B);

static LineMonitor     *gLineMonitor;
static Balancer        *gBalancer;
static BalancingWalker *gBalancingWalker;
static LineTracer      *gLineTracer;

static void user_system_create() {
  gBalancer = new Balancer();
  gBalancingWalker
    = new BalancingWalker(
      gGyroSensor, gLeftWheel,
      gRightWheel, gBalancer);
  gLineMonitor
    = new LineMonitor(gColorSensor);
  gLineTracer
    = new LineTracer(
      gLineMonitor, gBalancingWalker);

  ev3_led_set_color(LED_ORANGE);
}

// 続く
```

各クラスのインスタンスを作成し、メンバ変数に代入することで、オブジェクト間の関係づけ（リンク）を作成します

4-7. オブジェクト図を実装する（4）

EV3

実装

- オブジェクト図とソースコードの関係を確認しましょう

app.cpp

```
// 続き

static void user_system_destroy() {
  gLeftWheel.reset();
  gRightWheel.reset();

  delete gLineTracer;
  delete gLineMonitor;
  delete gBalancingWalker;
  delete gBalancer;
}

void ev3_cyc_tracer(intptr_t exinf) {
  act_tsk(TRACER_TASK);
}
```

周期ハンドラの中で
TRACER_TASKを起動すると、
C言語側のtracer_task関数が起動されます

```
void main_task(intptr_t unused) {
  user_system_create();
  ev3_sta_cyc(EV3_CYC_TRACER);
  slp_tsk();

  ev3_stp_cyc(EV3_CYC_TRACER);
  user_system_destroy();
  ext_tsk();
}

void tracer_task(intptr_t exinf) {
  if (ev3_button_is_pressed(BACK_BUTTON)) {
    wup_tsk(MAIN_TASK);
  } else {
    gLineTracer->run();
  }
    ext_tsk();
}
```

サンプルのソースコードを開いて、オブジェクト図とソースコードの対応関係を確認しましょう

4-8. タスクの処理を確認する（1）

EV3 ET ROBOT CONTEST

実装

- 周期ハンドラを実装します
  - app.cfgに、タスクと周期ハンドラの設定を追加します

app.cfg

```
INCLUDE("app_common.cfg");
#include "app.h"

DOMAIN(TDOM_APP) {
CRE_TSK(MAIN_TASK,  {TA_ACT,0,main_task,MAIN_PRIORITY,STACK_SIZE,NULL});
CRE_TSK(TRACER_TASK,{TA_NULL,0,tracer_task,TRACER_PRIORITY,STACK_SIZE,NULL});

EV3_CRE_CYC(EV3_CYC_TRACER, {TA_NULL, 0, ev3_cyc_tracer, 4, 1});
}
// ...
```

ここでRTOS側のMAIN_TASKがC言語側のmain_taskに関連づけられます

EV3RTの周期ハンドラは、EV3RT用に用意されたので、HRP2の仕様とは少し異なり、名称も別です

周期ハンドラを、自動起動しないときはTA_NULLに設定にします

周期ハンドラの起動周期は4msです

EV3RTにおける周期的なタスクの使い方について
下記ページに作成方法のサンプルと説明があります
http://dev.toppers.jp/trac_user/ev3pf/wiki/FAQ#FAQ

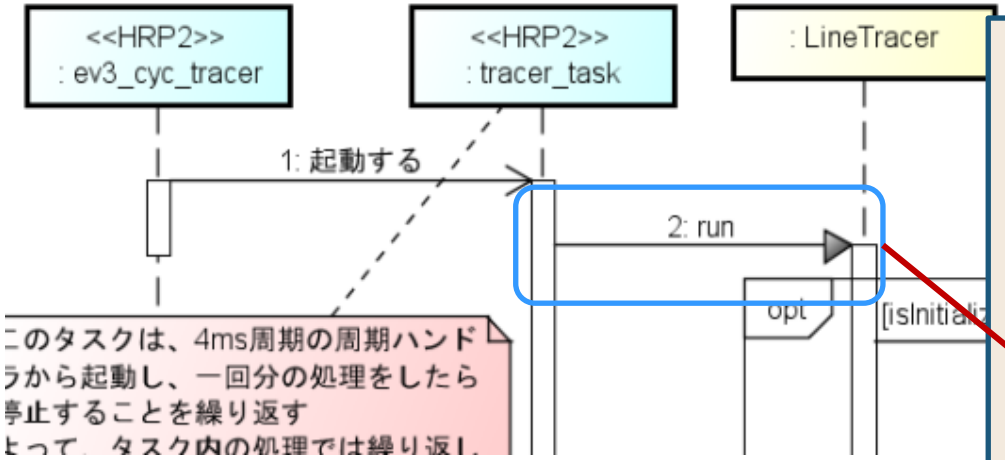
スライド中のコードは端折っていますので、詳細は、実際のソースコードで確認しましょう

4-8. タスクの処理を確認する（2）

EV3 ET ROBOT CONTEST

実装

- tracer_taskの処理の説明
  - アプリケーションのタスクを4msの周期ハンドラから呼び出されるように決めたので、tracer_taskはOSから繰り返し呼びだされます
  - タスクの処理に、LineTracer::run()を呼び出すコードを追加します



シーケンス図の一部（EV3RTの場合）

app.cpp

```
// ...
void tracer_task(intptr_t exinf) {
  if (ev3_button_is_pressed(...)) {
    wup_tsk(MAIN_TASK);
  } else {
    gLineTracer->run();
  }
  ext_tsk();
}
```

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう

スライド中のコードは端折っていますので、詳細は、実際のソースコードで確認しましょう

4-8. タスクの処理を確認する（3）

EV3 ET ROBOT CONTEST

実装

main_taskの処理の説明

- 動的に生成するインスタンスとリンクを作成します
- 周期ハンドラを起動します
- 終了時に周期ハンドラを停止し、インスタンスを破棄します

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう

app.cpp

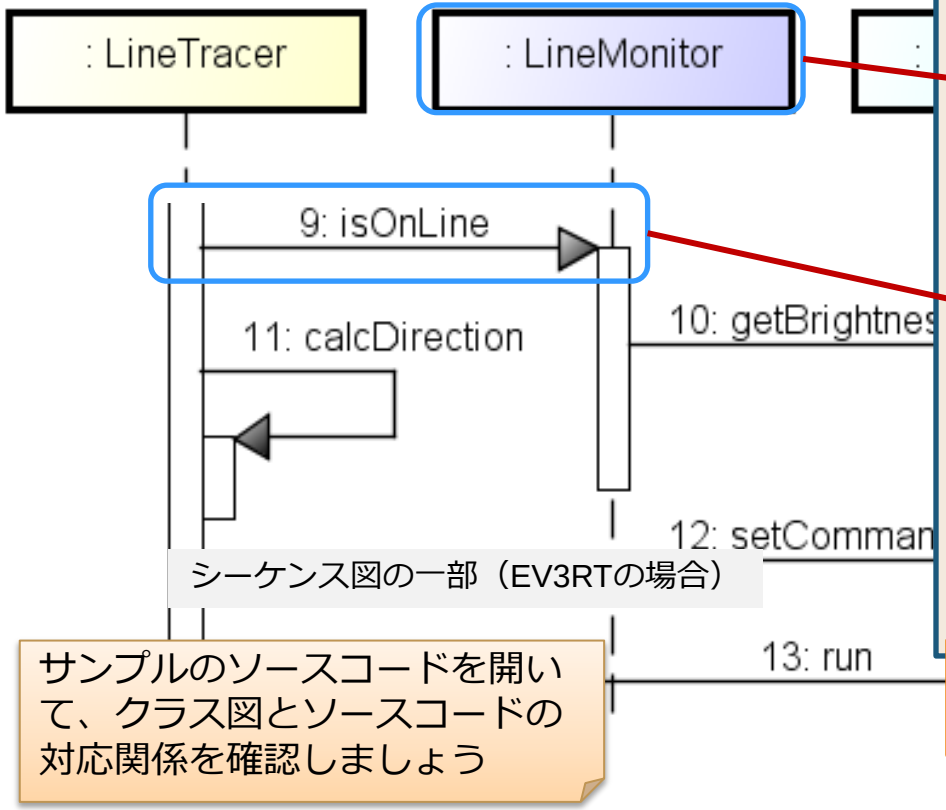
```
void main_task(intptr_t unused) {
  user_system_create();  // 動的に生成するインスタンスの初期化

  // 周期ハンドラ開始
  ev3_sta_cyc(EV3_CYC_TRACER);

  slp_tsk();  // バックボタンが押されるまで待つ

  // 周期ハンドラ停止
  ev3_stp_cyc(EV3_CYC_TRACER);

  user_system_destroy();  // 生成したオブジェクトの破棄
  ext_tsk();
}
```

4-9. シーケンス図のメッセージを確認する

実装

EV3

- シーケンス図のメッセージがクラスの操作になっていることを確認します
  - シーケンス図で、ライフラインオブジェクトが受け取っているメッセージに着目します
  - そのオブジェクトの属するクラスの操作にメッセージがあるかどうか調べます



シーケンス図の一部（EV3RTの場合）

```
unit/LineMonitor.h

class LineMonitor {
public:
  explicit LineMonitor(...);

  bool isOnLine() const;
  void setThreshold(int8_t threshold);

private:
    // ...
};
```

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう

スライド中のコードは端折っていますので、詳細は、実際のソースコードで確認しましょう

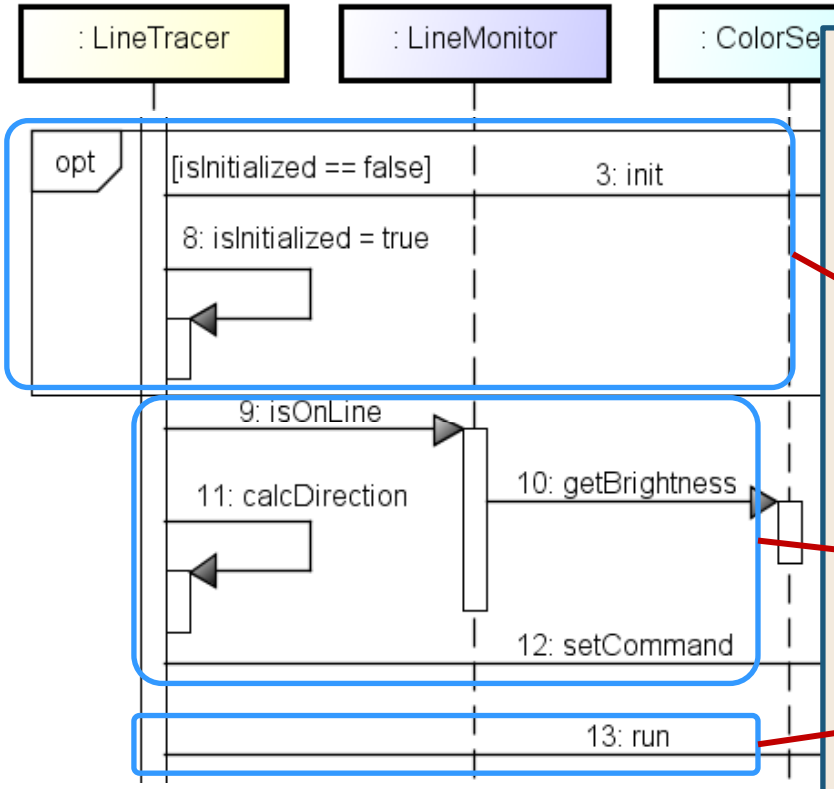
4-10. クラスの振舞いを実装する（1）

EV3 ET ROBOT CONTEST

実装

- LineTracer::run()メソッドを実装します

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう



app/LineTracer.cpp

```
void LineTracer::run() {
  if (mIsInitialized == false) {
    mBalancingWalker->init();
    mIsInitialized = true;
  }

  bool isOnLine = mLineMonitor->isOnLine();
  int direction = calcDirection(isOnLine);
  mBalancingWalker->setCommand(
    BalancingWalker::LOW, direction);

  mBalancingWalker->run();
}
```

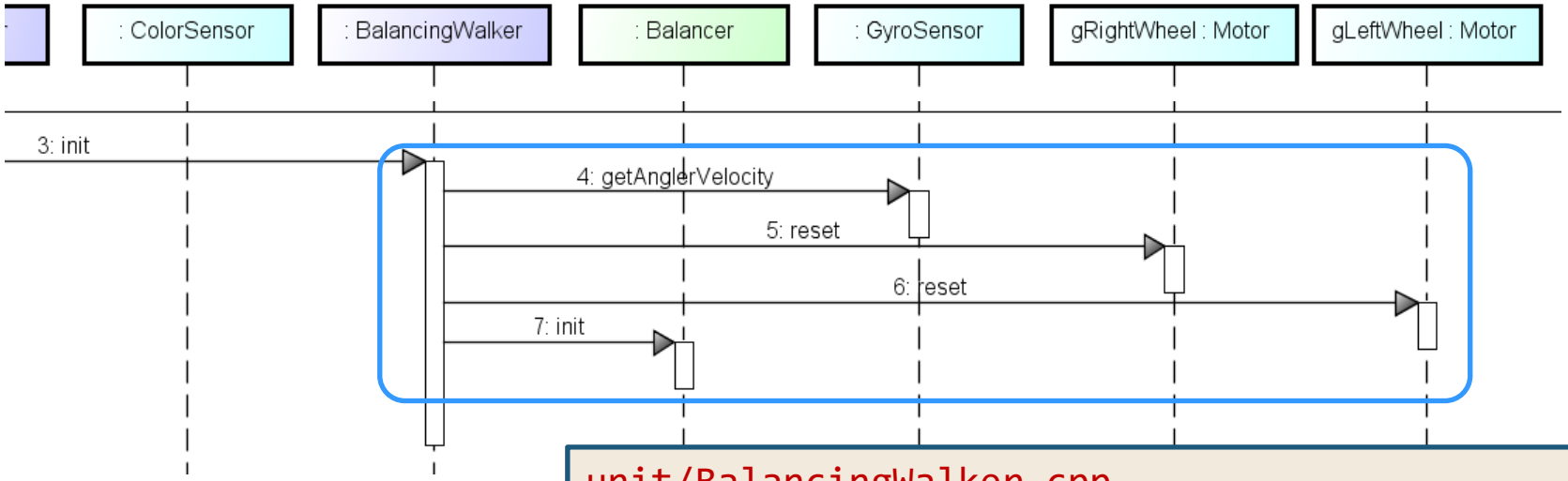
シーケンス図の一部（EV3RTの場合）

4-10. クラスの振舞いを実装する（2）

EV3

実装

- BalancingWalker::init()メソッドを実装します



シーケンス図の一部（EV3RTの場合）

unit/BalancingWalker.cpp

```
void BalancingWalker::init() {
  int offset = mGyroSensor.getAnglerVelocity();

  mLeftWheel.reset();
  mRightWheel.reset();

  mBalancer->init(offset);
}
```

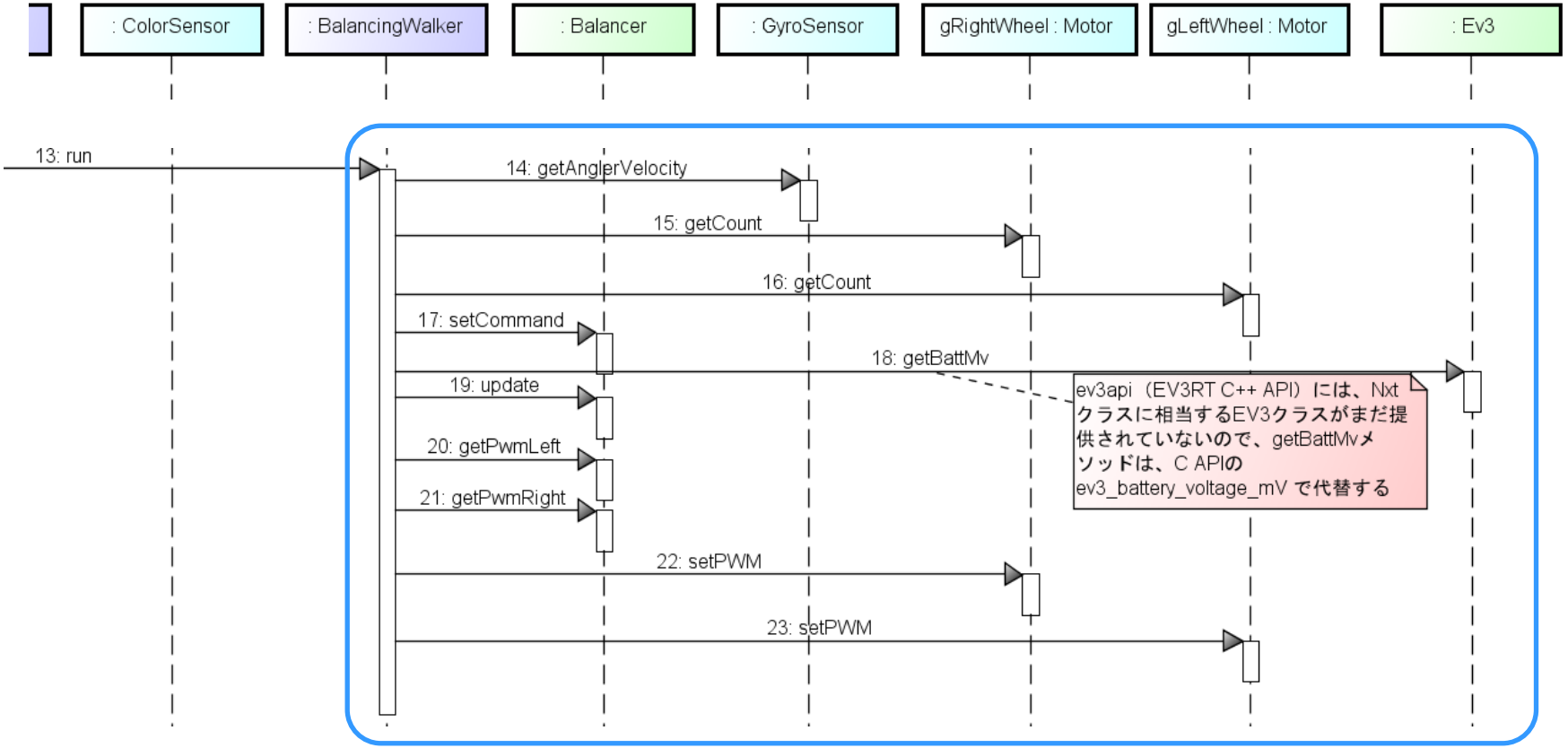
サンプルのソースコードを開いて、シーケンス図とソースコードの対応関係を確認しましょう

4-10. クラスの振舞いを実装する（3）

EV3

実装

- BalancingWalker::run()メソッドを実装します（続き



シーケンス図の一部（EV3RTの場合）

4-10. クラスの振舞いを実装する（4）

EV3 ET ROBOT CONTEST

実装

- BalancingWalker::run()メソッドを実装します（続き

```
unit/BalancingWalker.cpp

void BalancingWalker::run() {
  int16_t angle = mGyroSensor.getAnglerVelocity();
  int rightWheelEnc = mRightWheel.getCount();
  int leftWheelEnc  = mLeftWheel.getCount();

  mBalancer->setCommand(mForward, mTurn);

  int battery = ev3_battery_voltage_mV();
  mBalancer->update(
    angle, rightWheelEnc,
    leftWheelEnc, battery);

  mLeftWheel.setPWM(mBalancer->getPwmLeft());
  mRightWheel.setPWM(mBalancer->getPwmRight());
}
```

サンプルのソースコードを開いて、クラス図とソースコードの対応関係を確認しましょう

4-11. ソースコードの一覧

EV3 ET ROBOT CONTEST

実装

- この演習のソースコードは、次の構成になっています

```
etrobo_tr
├── Makefile.inc                  makeコマンドが使うMakefile
├── app                           アプリケーション
│   ├── LineTracer.cpp              ライントレーサcppファイル
│   └── LineTracer.h                ライントレーサヘッダファイル
├── app.cfg                       EV3RTのコンフィギュレーションファイル
├── app.cpp                       全体構成のためのcppファイル
├── app.h                         全体構成のためのヘッダファイル
├── balancer.c                    倒立振子制御APIソースファイル
├── balancer.h                    倒立振子制御APIヘッダファイル
├── balancer_param.c              倒立振子制御部が使うパラメータファイル
├── balancer_private.h            倒立振子制御部が使うヘッダファイル
└── unit                          ユニット
    ├── BalancerCpp.cpp             倒立振子制御クラスcppファイル
    ├── BalancerCpp.h               倒立振子制御クラスヘッダファイル
    ├── BalancingWalker.cpp         倒立走行部cppファイル
    ├── BalancingWalker.h           倒立走行部ヘッダファイル
    ├── LineMonitor.cpp             ライン監視部cppファイル
    └── LineMonitor.h               ライン監視部ヘッダファイル
```

ソースコード全体の構造（EV3RTの場合）

4-12. ソースコードをビルドする

EV3 ET ROBOT CONTEST

テスト

- makeコマンドを使ってソースコードをビルドします
  - `make app=アプリケーション名`
    - ファイルをコンパイルします（動的ローディング方式の場合のやり方）
  - `make clean`
    - `clean`を指示すると、中間オブジェクトファイルを削除します
- ビルドしてみましょう
  - `cd ev3rt/hrp2/sdk/beginners`
  - `make app=etrobo_tr`
  - 最後に「app」ファイルができていれば、ビルド成功です

4-13. 走行体を動かして動作を確認する

EV3

テスト

- 完成したプログラム(app)をMicroSDカードにコピーします
  - PCと電源を入れたEV3本体をUSBケーブルで接続します
  - 挿入してあるMicroSDカードがドライブとして認識されたら、作成したアプリケーション（app）をMicroSDカードの「ev3rt/apps/」ディレクトリにコピーします
  - コピーが終わったら、接続を解除し、USBケーブルを取り外します
- EV3の電源を入れ、転送したプログラムを選択します
  - EV3本体の中央ボタンを押し、電源を入れます
  - 下ボタンで、「SD card」を選択し、中央ボタンを押します。
  - 「app」を選択し、中央ボタンを押します。
- プログラムをスタートさせて、走行体がライントレースするかどうかを実際に確認してみましょう
  - LCD画面に「EV3way...」が表示されると走行体がスタートします
  - 停止には戻るボタンを押します

①中央ボタン　②左ボタン　③右ボタン
④下ボタン　⑤上ボタン　⑥戻るボタン

理解度チェック（４）

- モデルからソースコードを作る

1. 実装するためのモデルは、分析や設計のためのモデルと何が大きく異なりますか？
2. 構造モデルで表現されたものを、どのようにソースコードへ変換していましたか？
   a. 構造の視点からプログラムの独立性を高めるために、モデル実装時に考慮する点は何でしょうか？例えばc++言語ではどうしますか？
3. 振る舞いモデルで表現されたものを、どのようにソースコードへ変換していましたか？
4. 振る舞いの視点からプログラムの独立性を高めるために、モデル実装時に考慮する点は何でしょうか？