- これまで開発に使った部品の一部を再利用して、別の機能を持つロボットを開発します
- モデルを使った差分開発を体験してみましょう

6. 演習2）異なる機能を持つロボットを開発する

6-1. 演習課題と進め方

- 作ったロボットの部品と新たに開発する部品を使って、次の機能を持つロボットを作りましょう
  - 予め決められたシナリオに従って、倒立走行を行う「シナリオトレーサ」を開発します
    - 走行体は、ライントレーサと同様のものを使います
    - 経路に沿って走行するのではなく、自分が決めた時間と向きに走行します
- 演習1と同様に「モデルと部品を使った開発」の流れに沿って開発します
  - モデルを作成します
  - ソースコードを実装します
  - 実機での動作を確認します

6-2. 演習の準備（１）

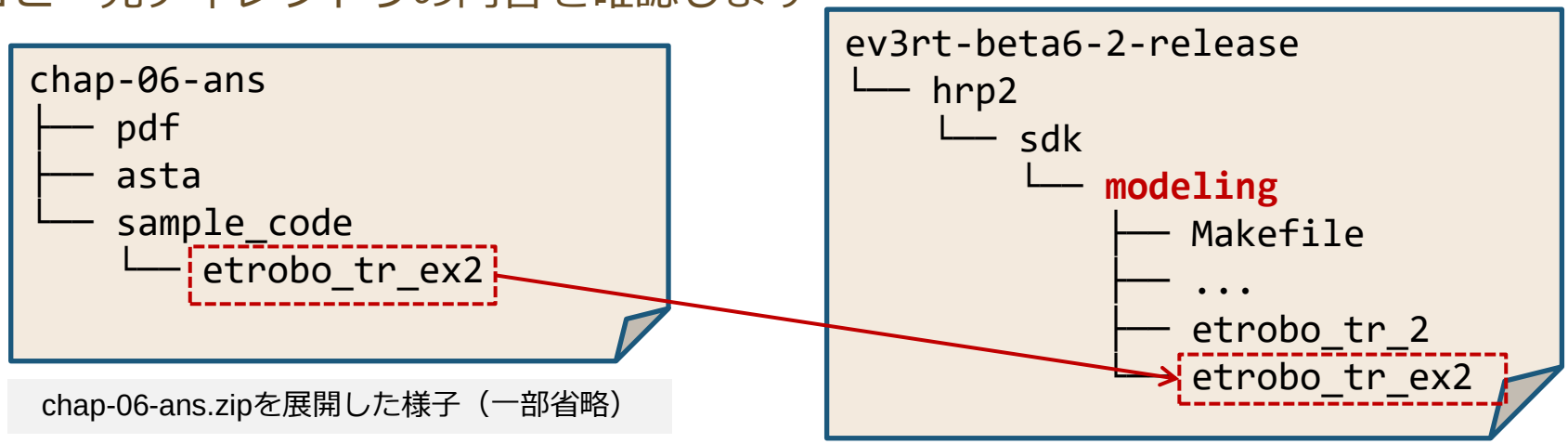
- 6章の解凍用パスワードを取得します
  - 圧縮ファイル解凍用のパスワードを取得してください
- 6章の演習後に使う配布ファイル
  - ファイル名：chap-06-ans.zip
  - 6章の演習のモデルとコードの解答例が含まれています
- 圧縮ファイルchap-06-ans.zipを展開します
  - 展開したディレクトリの中身を確認しましょう

```
chap-06-ans
├── pdf                          テキストのディレクトリ
│   └── chap-06-opened.pdf           6章のテキスト（解答例つき）
├── asta                         モデル図のディレクトリ
│   └── etrobo_tr_ex2.asta           演習2のサンプルモデル
└── sample_code                  サンプルコードのディレクトリ
    └── etrobo_tr_ex2/*              演習2のサンプルコード
```

「etrobo_tr_2」に対応した解答例が
「etrobo_tr_ex2」です

6-2. 演習の準備（2）

- モデル図の確認
  - 展開したディレクトリの「asta」ディレクトリにある「etrobo_tr_ex2.asta」が解答例のモデル図です
- ソースコードの確認
  - 展開したディレクトリの「sample_code」ディレクトリにある「etrobo_tr_ex2」が解答例のコードです
  - EV3RTをインストールしたディレクトリの中に「hrp2¥sdk」の中の「modeling」ディレクトリの中へ「etrobo_tr_ex2」をコピーします
  - コピー先ディレクトリの内容を確認します



chap-06-ans.zipを展開した様子（一部省略）

6-3. シナリオトレースの動作（１）

- 次の手順に従って走行するロボットを作ります

1. スタートボタンが押されるまで待つ
2. デバイスやライブラリを初期化する
3. シナリオからシーンを取り出す
   a. シナリオから、シーン番号順にシーンを1区間分読み込む
      - シーンは、1区間分の走行する時間（指定時間）と向き（コマンド）を示した動作仕様
4. シーンに合うよう倒立走行する
   a. 倒立走行に関する一連の手順を実行する
5. 走行時間が指定時間を超えたら、手順3へ戻り、次のシーンへ切り替える

6-3. シナリオトレースの動作（2）

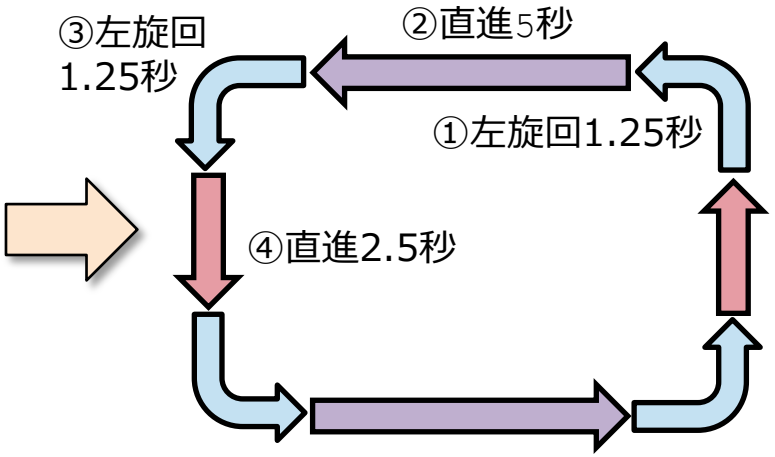
シナリオの表し方

- シナリオは、シーンの集まったものです
- シーンが備える情報は次の3種類です
  - シーン番号、走行する向き（コマンド）、指定時間
- シーンで使える走行する向きのコマンドは3種類あります
  - 直進する、右旋回する、左旋回する、静止する（倒立したまま停止する）

シナリオトレーサの動作の例

- 左表のようなシナリオだと、右図のような軌道を描くように走行します

シーン番号	走行する向き（コマンド）	指定時間
1	左旋回する	1.25秒
2	直進する	5秒
3	左旋回する	1.25秒
4	直進する	2.5秒

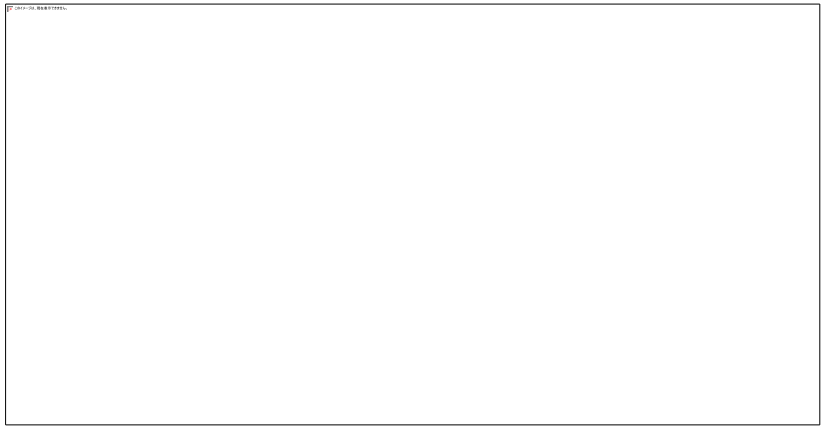


シナリオトレーサの実行するシナリオと動作の例

6-4. システムが提供する機能を決める（１）

要求分析

- アクタはユーザのままです
- ユーザに提供する機能は変わりますね
- 演習1のユースケース図を元に修正してみましょう
  - ユーザに提供する機能は演習1とは別のものになります
  - ユースケース名を「シナリオに沿って走行する」としましょう
- 作成したユースケース図は右のようになりました



シナリオトレーサの機能
（ユースケース図）

要求モデル・機能「スタータつきライントレーサの機能」を開いて、図の名前を「シナリオトレーサの機能」に変更し、ユースケース図を書き直しましょう（システムの名前が変わっています）

6-4. システムが提供する機能を決める（2）

- ユースケースの記述を要求に合わせて記述してみましょう
- ユースケース記述も見なおしてみましょう
  - ユーザに機能を提供するための手順を整理します
  - 一部の動作は、演習1で作成したものが使えそうです
  - シナリオを使うための手順や動作は新しく作成する必要があります
- 作成したユースケース記述は、右のようになりました

要求モデル・機能「シナリオトレーサの機能」を開いて、ユースケース図に書いてあるユースケース記述を書き直しましょう

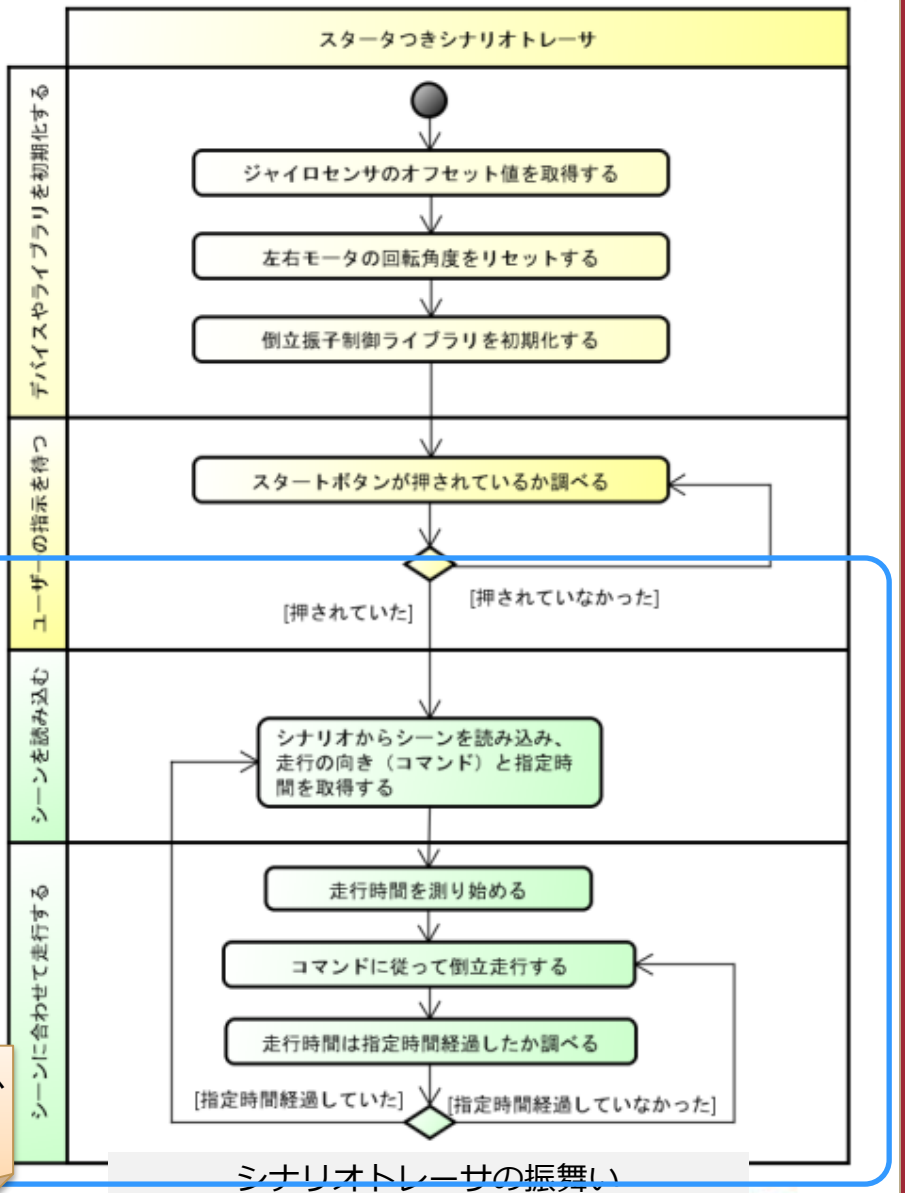
要求分析

ユースケース「シナリオに
する」の基本ユースケース

1. ユーザがロボットを起動すると、ロボットは動作を開始する
2. ロボットは動作を開始すると、デバイスやライブラリを初期化し、走行開始の指示を待つ
3. ユーザは、スタートボタン（タッチセンサ）を押してロボットに走行開始を指示する
4. ロボットは、ボタンが押されると倒立走行する
5. ロボットは、シナリオからシーンを読み込む
6. ロボットは、シーンにある指定時間と向きに従って倒立走行する
7. システム時刻を使って指定時間が経過したか調べ、次のシーンに進む
8. 手順5～7を繰り返す

6-5. 機能の実現方法を考える

- 演習1のアクティビティ図を元に修正してみましょう
  - スタートボタンを押すところまでは、演習1と同じです
  - シナリオからシーンを取り出すところを追加します
  - 時間が経過したらシーンを読み直すところを追加します
- 上記を反映したアクティビティ図は右のようになります
  - 水色部分が新しく変更した箇所です
  - 倒立走行は、演習1と同じようにライントレーサクラスに任せるので、詳細は省略しています



シナリオトレーサの振舞い
（アクティビティ図）

要求モデル・振舞い「経路に沿って走行する」を開いて、図の名前を「シナリオに沿って走行する」に変更し、アクティビティ図を書き直しましょう

6-6. 機能実現に必要な部品を見つける（１）

基本設計

- 部品の候補を考えて表を作ってみましょう
  - 「シーン」が集まった「シナリオ」がある
  - シーンは、「指定時間」「コマンド（走行の向き）」を保持している
  - シナリオからシーンを読み込む、次のシーンに進む
  - シーンに含まれるコマンドで指定時間走行する
  - 指定時間経過したことを調べる
  - 倒立走行部や、演習1で作成したスタータが使えそう
  - 例題1のライントレーサを部品として使ってスターターつきのライントレーサのように、この演習でもシナリオに沿って走行する部品と、それを使うスタータつきの部品に分けることができそう

6-6. 機能実現に必要な部品を見つける（２）

基本設計

- 新しく作った表は、下のような表になりました

「働き」や「情報」	部品の候補
シナリオからシーンを読み込む	シナリオ、シーン
次のシーンに進む	シナリオ、シーン
シーンに合わせて走行する	シナリオトレーサ
指定時間経過したことを調べる	経過時間を計測する（タイマー）
システム時刻を調べる	システム（EV3の本体やOS）の時計
倒立走行する	倒立走行部
スタート機能	スタータ
スタートしたらシナリオに沿って走行する	スタータつきのシナリオトレーサ

シナリオトレーサの働きや部品の候補

6-7. 新しい部品を定義する（1）

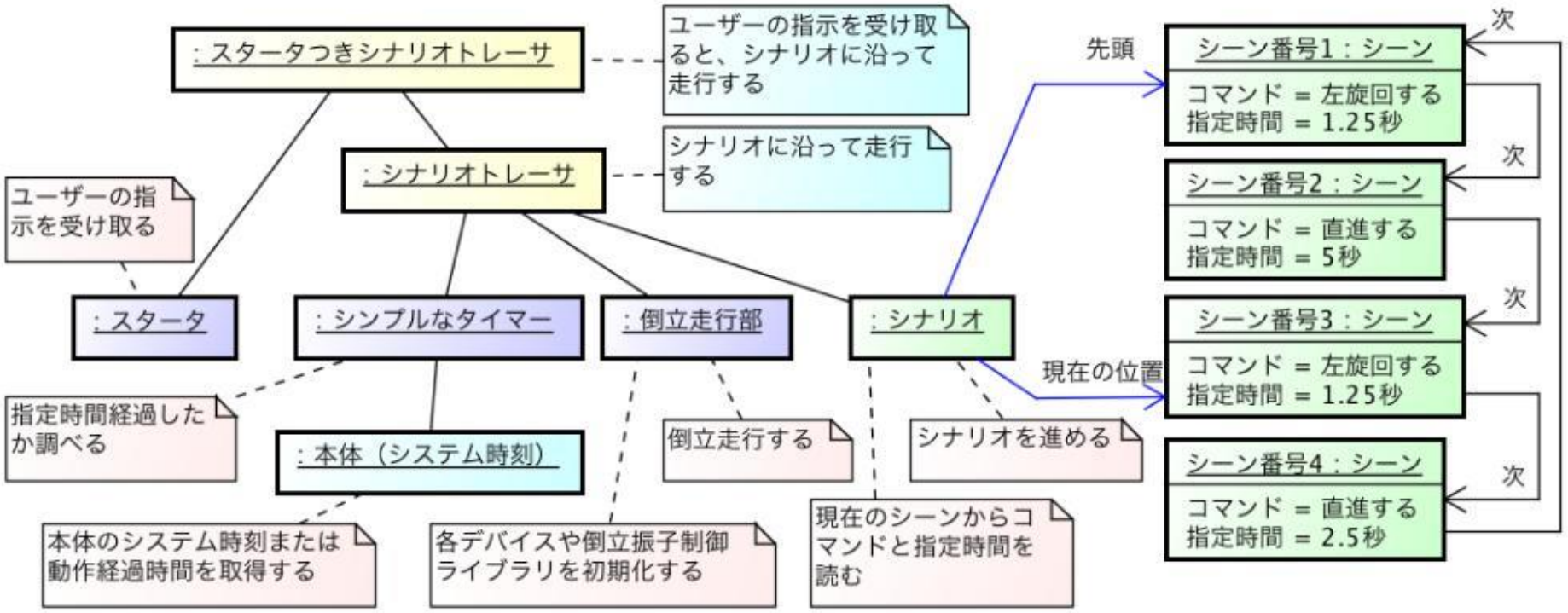
基本設計

- 演習1のスタートつきライントレーサの使わないオブジェクトを削除します
  - スタータと倒立走行部は詳細を隠します
    - 残しておいたほうが分かりやすい場合は消さなくてもよいです
- 「シナリオトレース機能」を実現するために必要な部品を追加します
  - シナリオトレーサ、シナリオ、シーン、タイマーを追加しましょう

設計モデル・構造「スタータつきライントレーサの部品の候補のつながり」を開いて、図の名前を「スタータつきシナリオトレーサの部品の候補のつながり」に変更し、オブジェクト図を書き直しましょう

6-7. 新しい部品を定義する（2）

- 新しく作ったオブジェクト図は、下のような図になりました



スタータつきシナリオトレーサの部品の候補のつながり
（オブジェクト図）

6-8. 部品による機能実現を確認する（１）

- 演習1のスタートつきライントレーサの使わないオブジェクトを削除します
  - スタータと倒立走行部は詳細を隠します
    - 残しておいたほうが分かりやすい場合は消さなくてもよいです

基本設計

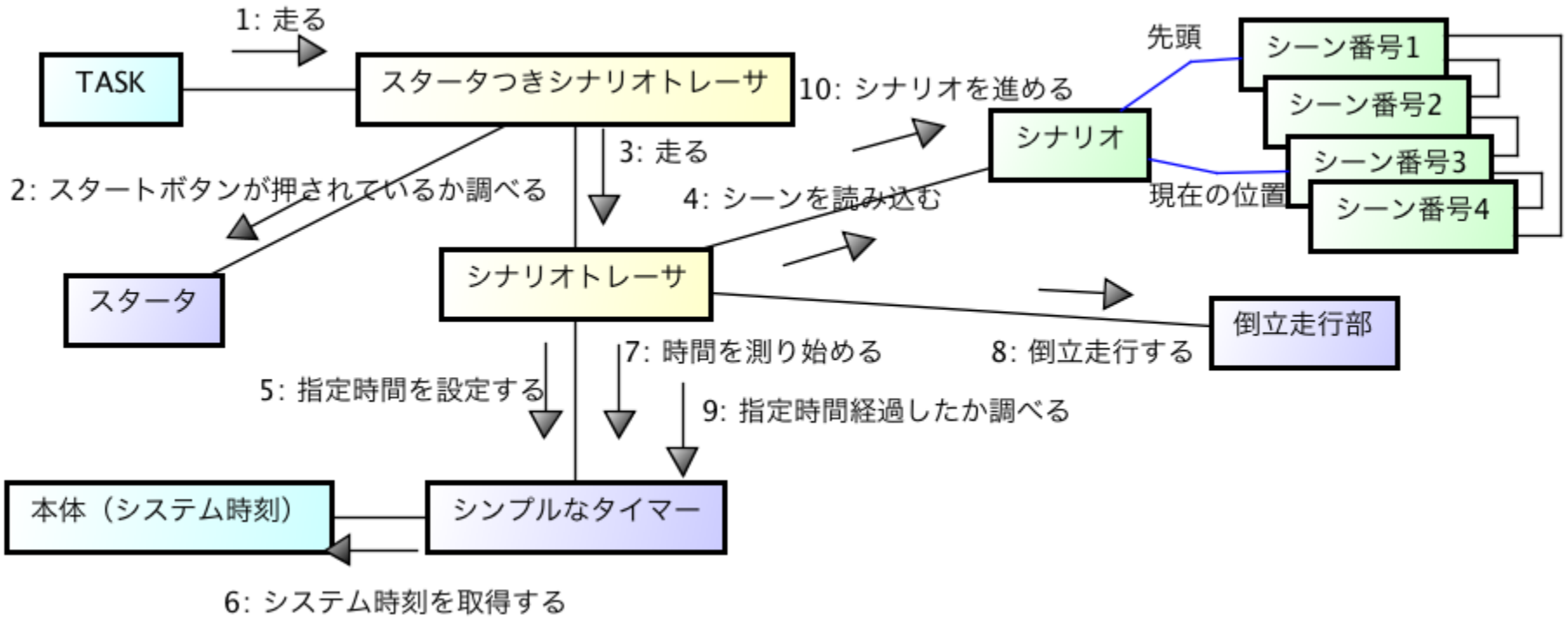
- 「シナリオトレース機能」を実現するのに必要なやりとりを書きます
  - シナリオトレーサ、シナリオ、シーン、タイマーを追加します
  - 候補を探したときに得た働きを参考にやりとりに使うメッセージを追加しましょう

設計モデル・振舞い「スタータつきライントレーサの部品どうしのやりとり」を開いて、図の名前を「スタータつきシナリオトレーサ」に変更し、コミュニケーション図を書き直しましょう

6-8. 部品による機能実現を確認する（２）

- 新しく作ったコミュニケーション図は、下のような図になりました

基本設計



シナリオトレーサの部品どうしのやりとり
（コミュニケーション図）

6-9. システムの構造を決定する

基本設計

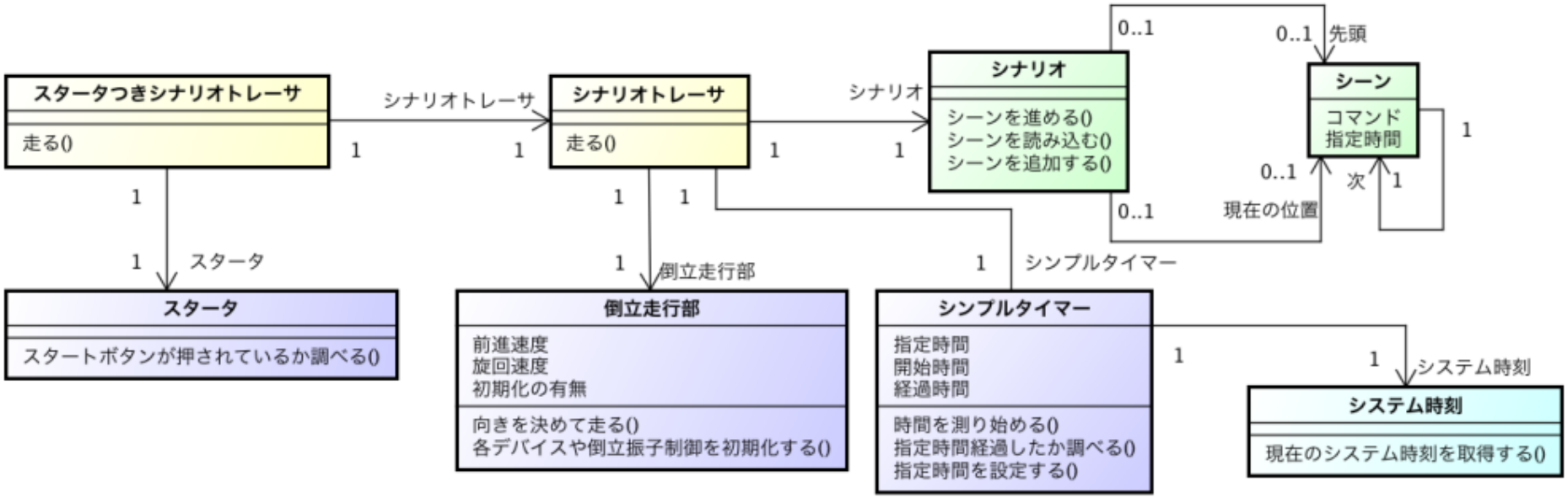
- 演習1のスタートつきライントレーサの使わないクラスを削除します
  - スタータと倒立走行部は詳細を隠します
    - 残しておいたほうが分かりやすい人は残しておいても構いません
- 「シナリオトレース機能」を実現するのに必要なやりとりを書きます
  - シナリオトレーサ、シナリオ、シーン、タイマーを追加します
  - コミュニケーション図で使ったやりとりを操作に追加しましょう

設計モデル・構造「スタータつきライントレーサの構造」を開いて、図の名前を「スタータつきシナリオトレーサの構造」に変更し、クラス図を書き直しましょう

6-9. システムの構造を決定する

基本設計

- 新しく作ったクラス図は、下のような図になりました
  - シーンから次のシーンへのつながりは、シナリオクラスの「シーンを追加する」メソッドがシーンを追加するときに作成します
  - シーンは「次」の要素で繋がる単方向リストとします
    - シーンの追加：シナリオクラスが担当、最後の要素と先頭の要素の間に挿入します

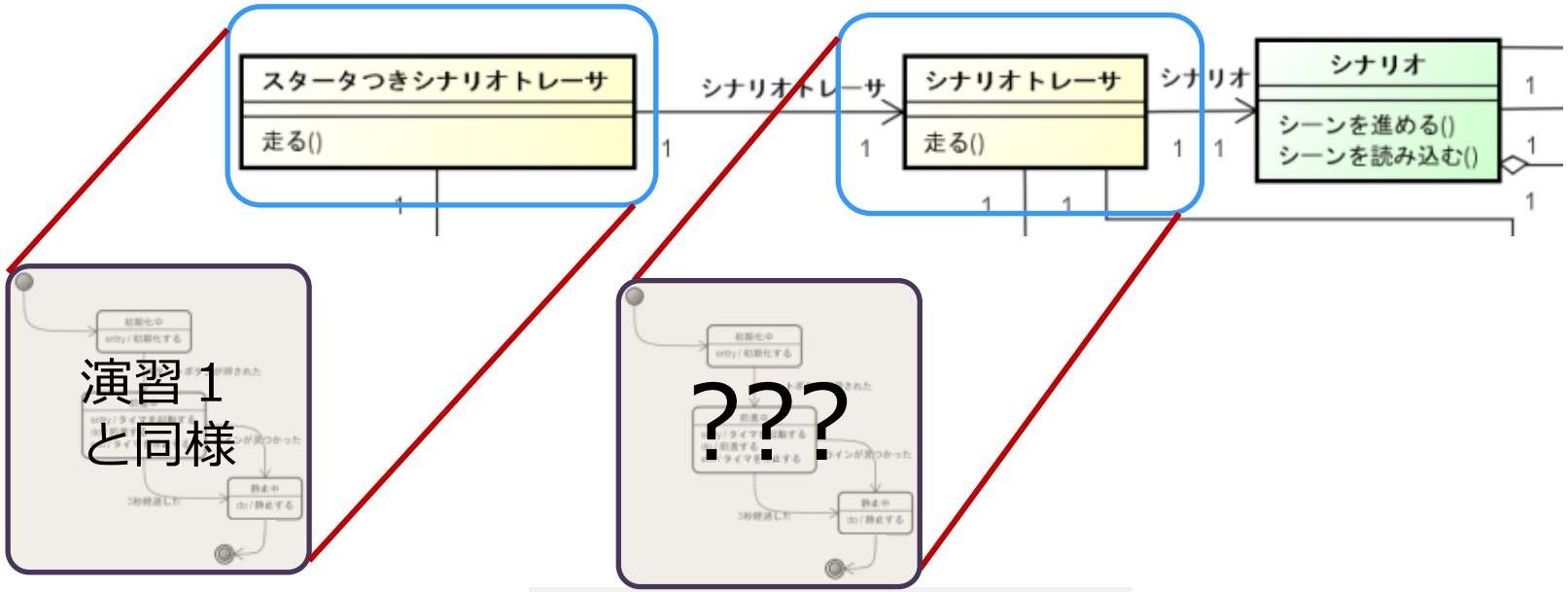


スタータつきシナリオトレーサの構造
（クラス図）

6-10. システムの振舞いを決定する（１）

基本設計

- スタータつきシナリオトレーサクラスの振舞い
  - 演習1のスタータつきライントレーサクラスとほぼ同じですので省略します
- シナリオトレーサクラスのステートマシン図を書いてみましょう
  - 6-6で作成した表などを参考に、起きるのを待っているできごとを探して、状態の候補を見つけましょう



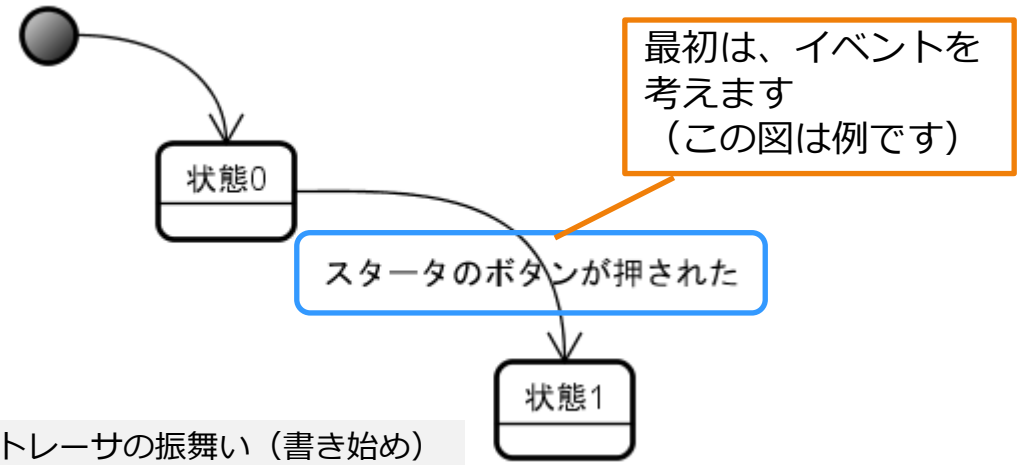
シナリオトレーサクラスの振舞いを作る
（クラス図とステートマシン図）

6-10. システムの振舞いを決定する（２）

基本設計

- ステートマシン図を追加します
  - 開始擬似状態と最初の状態を追加して、状態遷移でつなぎます
  - ２つ目の状態を追加します
- 状態遷移とイベントを書きます
  - ライントレーサは、最初の「待っているできごと」が何かを考えます
  - 最初の状態と２つ目の状態の間に状態遷移を追加します
  - 最初の待っているできごとをこの状態遷移のイベントとします
    - イベントにするときは、起きてほしいできごとが「起きた」と言い表します

設計モデル・振舞い「スタータつきライントレーサの「走る」の振舞い」開き、図の名前を「シナリオトレーサの振舞い」に変更し、ステートマシン図を書き直しましょう

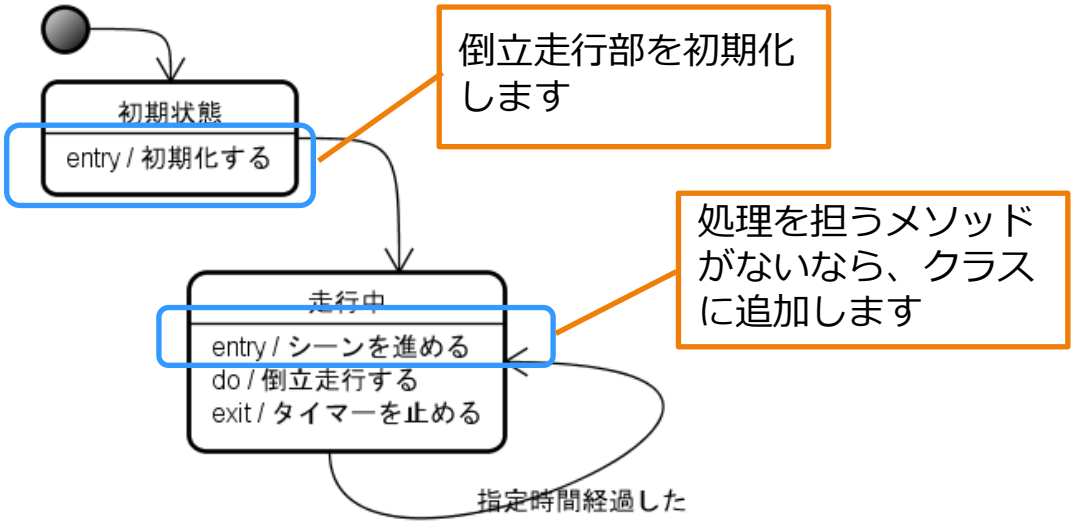


シナリオトレーサの振舞い（書き始め）
（ステートマシン図）

6-10. システムの振舞いを決定する（3）

基本設計

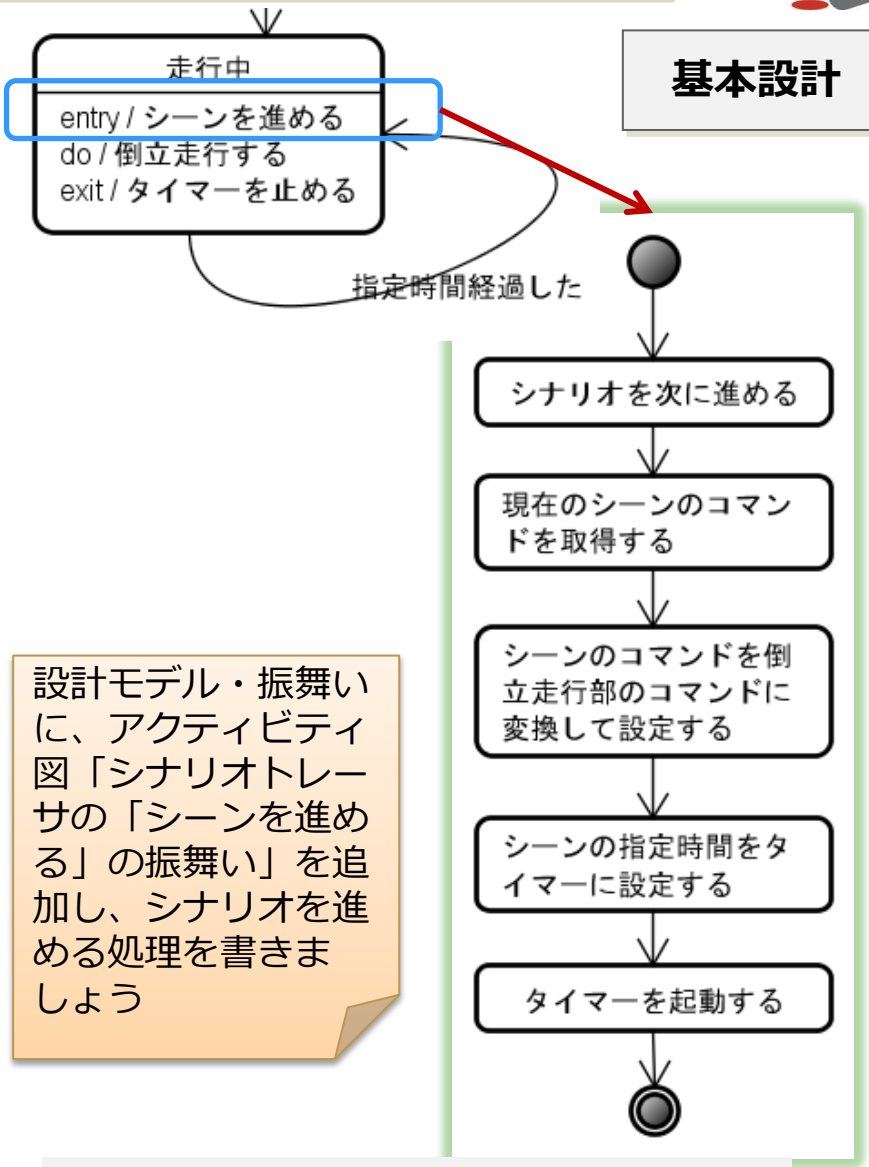
- アクションを書きます
  - それまでにやっておくことを最初の状態のアクションに書きます
    - 操作の呼び出しを「～をする」と書きます（なければアクションはなしです）
- 状態の名前を決めます
  - 起きるのをまっているできごとからつけるなら「～待ち」とつけます
  - できごとが起きるまでの振舞いから付けるなら「～中」とつけます
- 新しく作ったステートマシン図は下のような図になりました



シナリオトレーサの振舞い（アクションと状態名の追加後）（ステートマシン図）

6-10. システムの振舞いを決定する（4）

- 「シーンを進める」の処理をアクティビティ図で表しましょう
  - 状態の中のアクションが複雑であったり、複数の手続きであるときは、メソッドを追加してまとめます
  - 処理をまとめたいときは、状態をまとめるのではなく、それぞれの状態で同じメソッドと使うように整理しましょう
- 追加した走行中の「シーンを進める」のアクティビティ図は、右のような図になりました



シナリオトレーサの「シーンを進める」の振舞い（アクティビティ図）

6-11. 実装モデルに変換する

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（シナリオトレーサ）、unit（ライン監視部や倒立走行部）、platform（OSやEV3RT C++など）に分け、パッケージとして扱いましょう
- 今回使うタスクやハンドラの設定を決定します
  - 今回も、アプリケーション部分は1つのタスクで構成しましょう
  - アプリケーション内には繰り返し構造を持たせず、1周期分の処理を書きます
    - スタータつきシナリオトレーサが、ライントレーサ、シナリオトレーサの１周期分の処理を呼び出すようにして、倒立走行部の周期処理を維持しましょう
- 提供されたライブラリのクラスやAPIに読み替えます
  - シンプルタイマーが使うシステム時刻の取得には、Clockクラスを使います
- 自分たちで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
  - スタータつきシナリオトレーサ：ScenarioTracerWithStarter
    シナリオトレーサ：ScenarioTracer
    シナリオ：Scenario
    シーン：Scene
    シンプルタイマー：SimpleTimer
    などとします

6-12. 実装モデルを作成する（１）

- 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

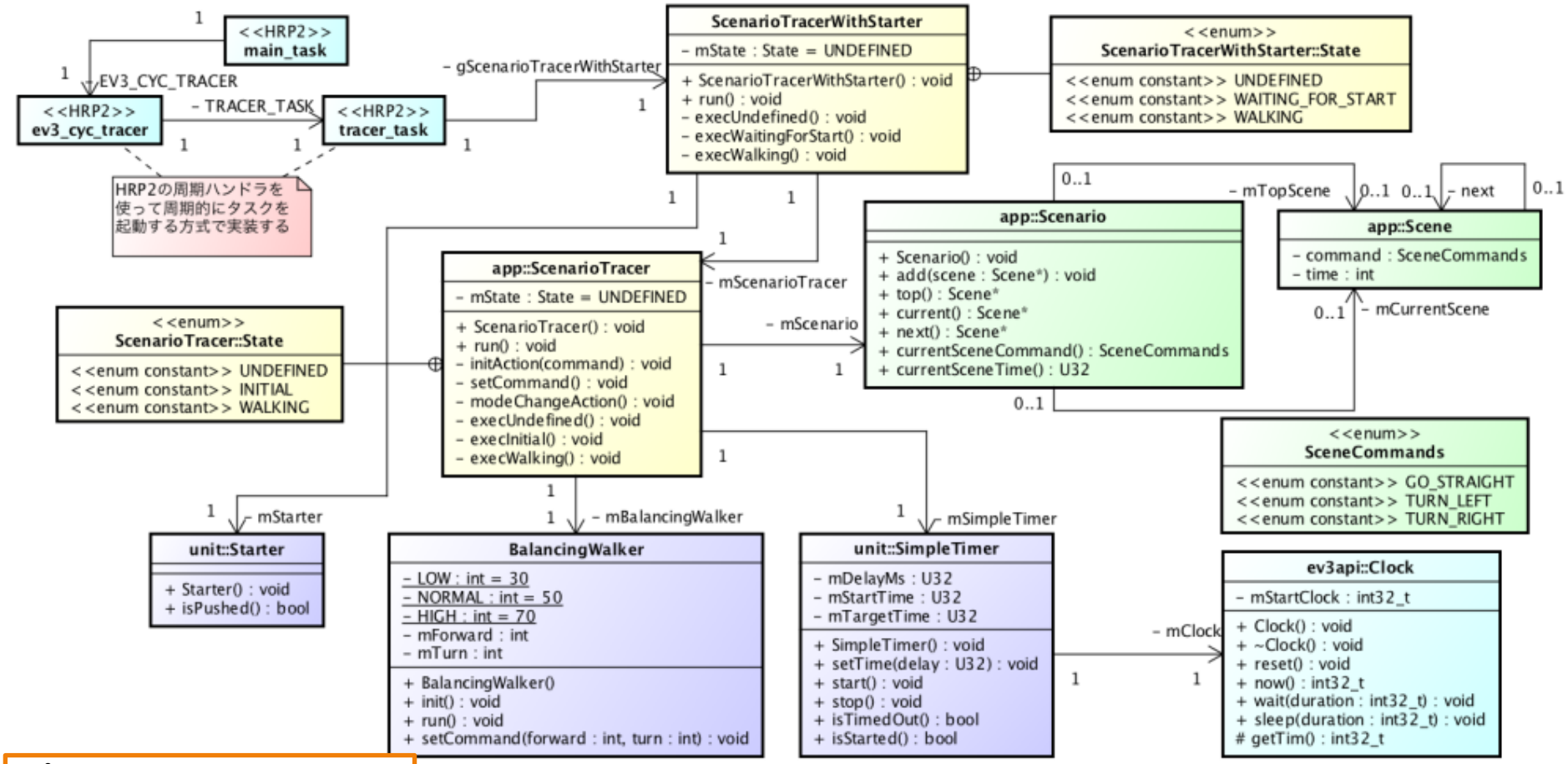
詳細設計

実装モデル・構造「スタータつきライントレーサの構造」を開いて、図の名前を「スタータつきシナリオトレーサの構造」に変更し、実装モデルのクラス図を書き直しましょう

6-12 実装モデルを作成する（2）

- 新しく作ったクラス図は下のような図になりました

詳細設計



パラメータの一部は非表示にしています

スタータつきライントレーサの構造（EV3RTの場合）
（クラス図）

6-12. 実装モデルを作成する（3）

EV3

詳細設計

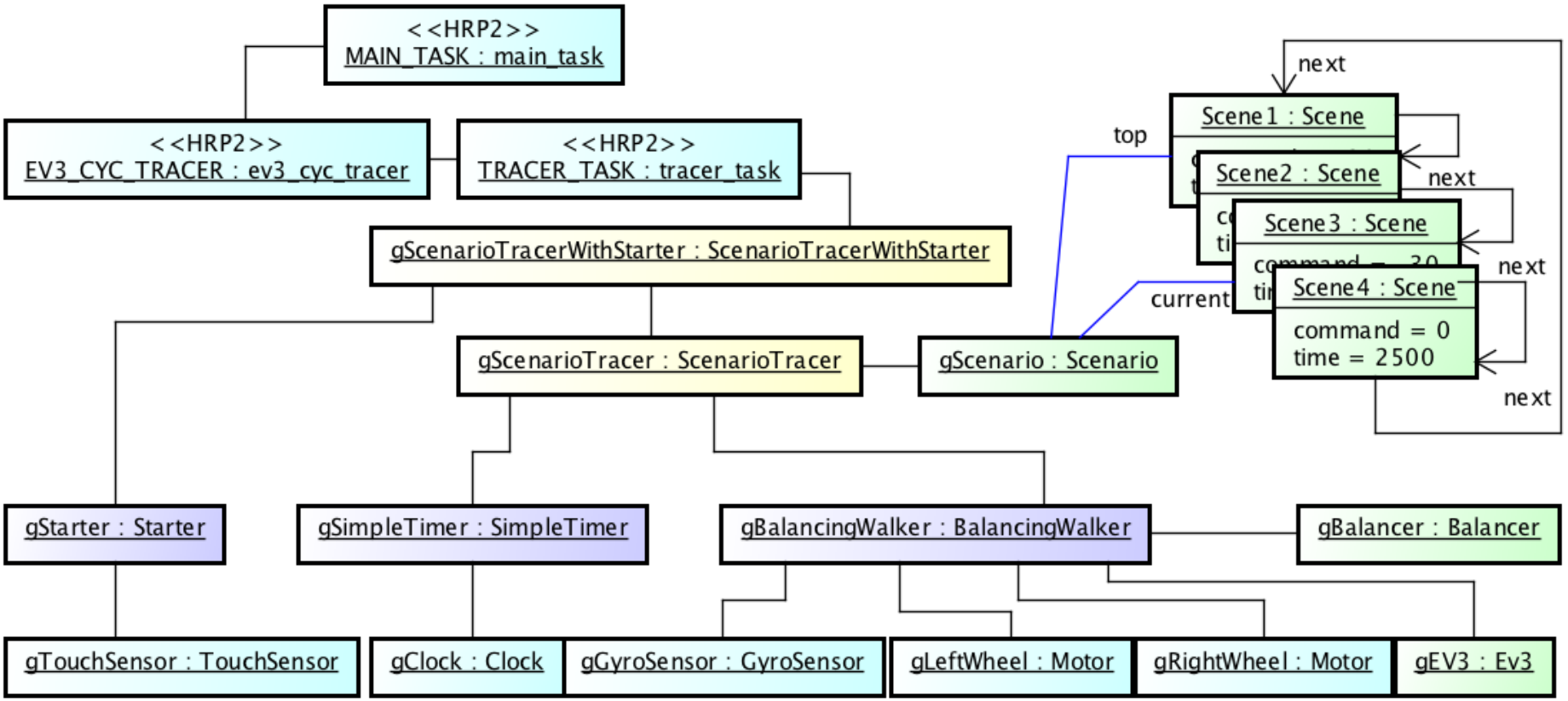
- 実装モデルのオブジェクト図を作成します
  - 設計モデルのオブジェクト図に登場することばを、実装モデルのクラス図で用意したことばで置き換えます
  - オブジェクト名は、実装するときにオブジェクトを作成するときに使う名前になります

実装モデル・構造「スタータつきライントレーサのオブジェクト」を開いて、図の名前を「スタータつきシナリオトレーサのオブジェクト」に変更し、実装モデルのオブジェクト図を書き直しましょう

6-12. 実装モデルを作成する（4）

- 新しく作ったオブジェクト図は下のような図になりました

詳細設計

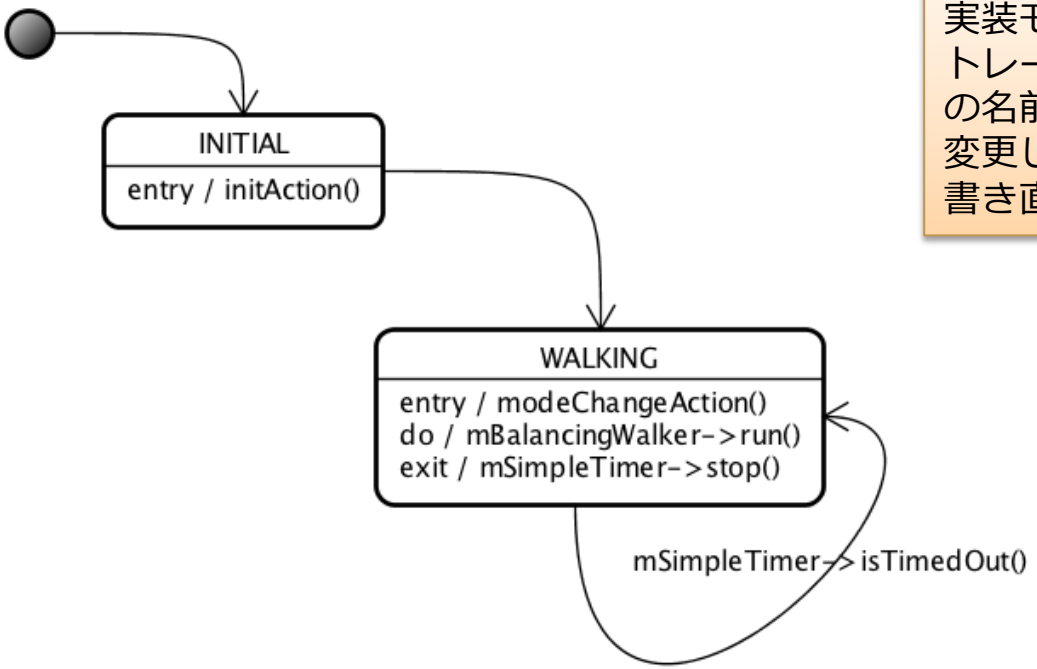


スタータつきライントレーサのオブジェクト（EV3RTの場合）
（オブジェクト図）

6-12. 実装モデルを作成する（５）

詳細設計

- 実装モデルのステートマシン図を作成します
  - 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
- 新しく作ったステートマシン図は、下のような図になりました



シナリオレーサの構造（EV3RTの場合）
（ステートマシン図）

実装モデル・振舞い「スタータつきラインレーサの「走る」の振舞い」を開き、図の名前を「シナリオトレーサの振舞い」に変更し、実装モデルのステートマシン図を書き直しましょう

6-13. ソースコードを書く（1）

実装

- 構造のアーキテクチャに従ってクラスを配置するパッケージを決定します
  - ScenarioTracerなどは「app」、SimpleTimerは「unit」に配置しましょう
- 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、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
- 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
- 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
- ヘッダファイルとcppファイルにコンストラクタとデストラクタを追加します

6-13. ソースコードを書く（2）

実装 EV3

複製したソースコードを開いて、変換ルールに従ってソースコードを追加、修正しましょう

- ヘッダファイルを作成しましょう
  - Scenarioの例

app/Scenario.h

```
#ifndef EV3_APP_SCENARIO_H_
#define EV3_APP_SCENARIO_H_

#include "ev3api.h"
#include "Scene.h"

class Scenario {
public:
  explicit Scenario(Scene* scene);

  void add(Scene* scene);
  Scene* top() const;
  Scene* current() const;
  Scene* next();
  SceneCommands
    currentSceneCommand() const;
  uint32_t currentSceneTime() const;

private:
  Scene* mTopScene;
  Scene* mCurrentScene;
};

#endif  // EV3_APP_SCENARIO_H_
```

他のクラスも同様に作成します

6-13. ソースコードを書く（3）

実装 EV3

複製したソースコードを開いて、変換ルールに従ってソースコードを追加、修正しましょう

- CPPファイルを作成しましょう
  - Scenarioの例

app/Scenario.cpp

```
#include "Scenario.h"

Scenario::Scenario(Scene* scene)
  : mTopScene(scene),
    mCurrentScene(scene) {
}

void Scenario::add(Scene* scene) {
  if (mTopScene == 0) {
    mTopScene = scene;
  } else {
    Scene* s = mTopScene;
    while (s->next != top()) {
      s = s->next;
    }
    s->next = scene;
  }
  scene->next = top();
}

Scene* Scenario::top() const {
  return mTopScene;
}

Scene* Scenario::current() const {
  return mCurrentScene;
}
```

他のクラスも同様に作成します

6-13. ソースコードを書く（4）

実装 EV3

複製したソースコードを開いて、変換ルールに従ってソースコードを追加、修正しましょう

- CPPファイルを作成しましょう
  - Scenarioの例

app/Scenario.cpp

```
// 続き

Scene* Scenario::next() {
  if (mCurrentScene != 0) {
    mCurrentScene = mCurrentScene->next;
  } else {
    mCurrentScene = mTopScene;
  }
  return mCurrentScene;
}

SceneCommands
Scenario::currentSceneCommand() const {
  return (mCurrentScene == 0) ?
    GO_STRAIGHT : mCurrentScene->command;
}
```

```
uint32_t Scenario::currentSceneTime() const
{
  return (mCurrentScene == 0) ?
      0 : mCurrentScene->time;
}
```

他のクラスも同様に作成します

6-14. クラスの振舞いを実装する（1）

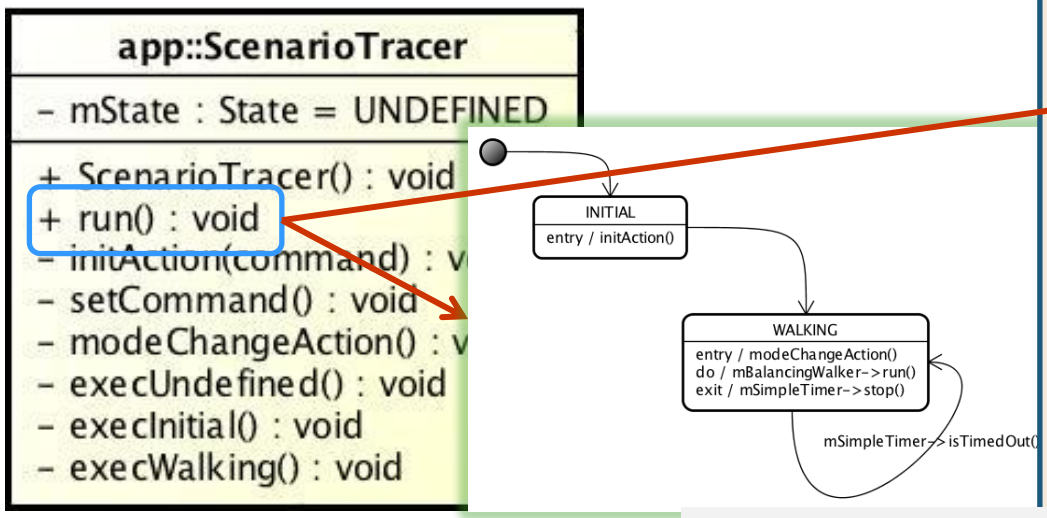
EV3

実装

- シナリオトレーサのrunメソッドをステートマシンを動かすメソッドとして実装しましょう
  - runメソッドは、周期ハンドラから繰り返し呼び出されることを思い出しましょう
  - UNDEFINEDはいずれの状態でもないことを表します

app/ScenarioTracer.cpp

```
// ...
void ScenarioTracer::run() {
    switch (mState) {
    case UNDEFINED:
        execUndefined();
        break;
    case INITIAL:
        execInitial();
        break;
    case WALKING:
        execWalking();
        break;
    default:
        break;
    }
}
```



シナリオトレーサの振舞いの実装（cppファイルの作成）

変換ルールに従ってソースコードを追加、修正しましょう

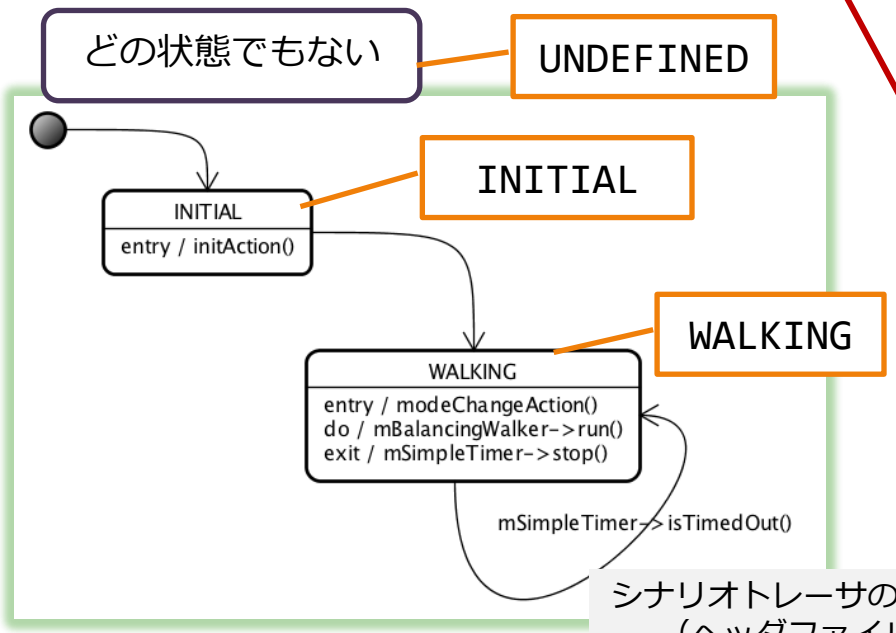
6-14. クラスの振舞いを実装する（2）

EV3

実装

- シナリオトレーサのヘッダファイルを編集しましょう
  - 状態を表すenumを定義します
  - 現在の状態を保持する属性を追加します
  - 状態ごとの処理用のメソッドを追加します

変換ルールに従ってソースコードを追加、修正しましょう



app/ScenarioTracer.h

```
class ScenarioTracer {
// ...
private:
  enum State {
    UNDEFINED,
    INITIAL,
    WALKING
  };
  // ...
  State mState;
  // ...
  void execUndefined();
  void execInitial();
  void execWalking();
};
```

シナリオトレーサの振舞いの実装（ヘッダファイルの作成）

6-14. クラスの振舞いを実装する（3）

EV3

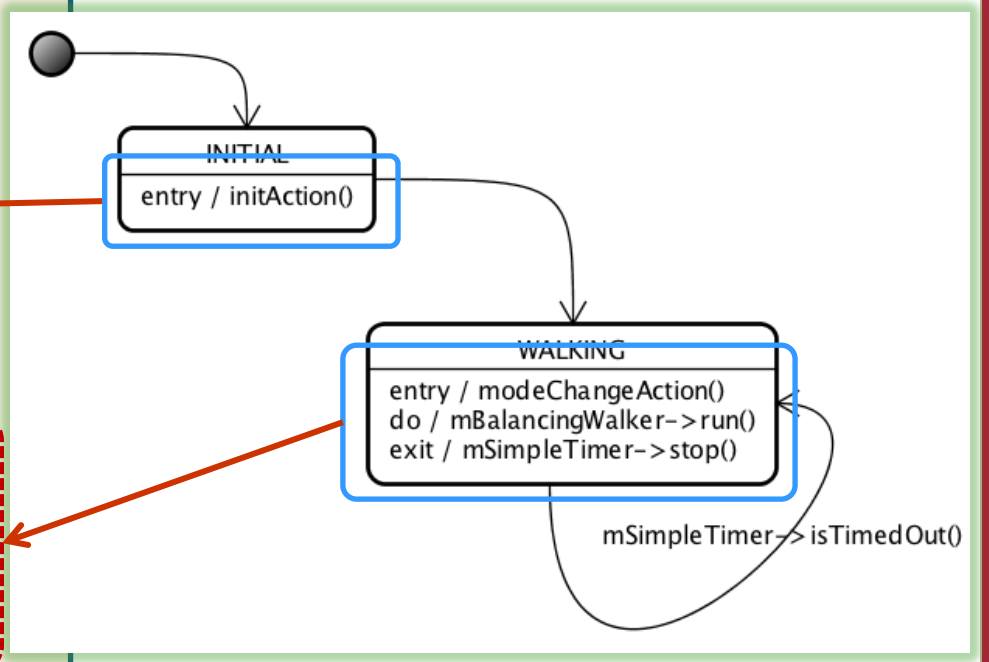
実装

- 状態ごとのメソッドをステートマシン図に合わせて編集しましょう

変換ルールに従ってソースコードを追加、修正しましょう

app/ScenarioTracer.cpp

```
// UNDEFINEDの処理
void ScenarioTracer::execUndefined() {
  mState = INITIAL;
}
// INITIALの処理
void ScenarioTracer::execInitial() {
  initAction();
  mState = WALKING;
  modeChangeAction();
}
// WALKINGの処理
void ScenarioTracer::execWalking() {
  mBalancingWalker->run();
  if (mSimpleTimer->isTimedOut()) {
    mSimpleTimer->stop();
    modeChangeAction();
  }
}
```



シナリオトレーサの振舞いの実装（cppファイルの作成）

6-14. クラスの振舞いを実装する（4）

EV3 ET ROBOT CONTEST

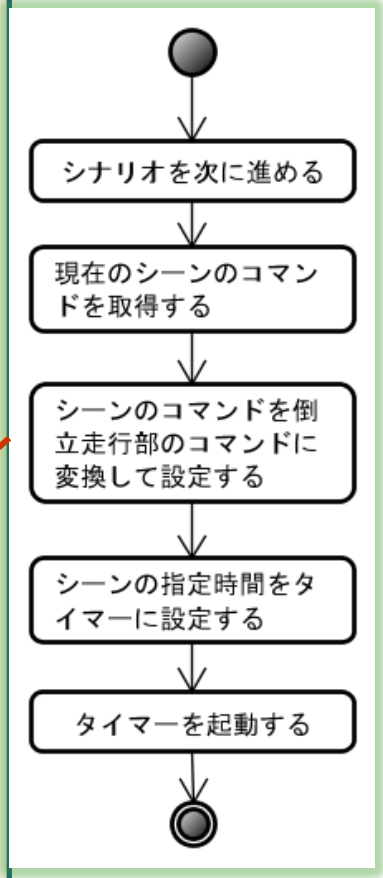
実装

- そのほかのメソッドも実装しましょう

変換ルールに従ってソースコードを追加、修正しましょう

app/ScenarioTracer.cpp

```
void ScenarioTracer::initAction() {
  mBalancingWalker->init();
}
void ScenarioTracer::setCommand(SceneCommands command) {
  int turn = 0;
  if (command == TURN_LEFT) {
    turn = -BalancingWalker::LOW;
  } else if (command == TURN_RIGHT) {
    turn = BalancingWalker::LOW;
  }
  mBalancingWalker->setCommand(BalancingWalker::LOW, turn);
}
void ScenarioTracer::modeChangeAction() {
  mScenario->next();
  SceneCommands command = mScenario->currentSceneCommand();
  setCommand(command);
  mSimpleTimer->setTime(mScenario->currentSceneTime());
  mSimpleTimer->start();
}
```



シナリオトレーサの振舞いの実装（cppファイルの作成）

6-15. ビルドし走行体を動かす（１）

EV3

テスト

- コピーした演習1のソースコードと今回新たに作成したソースコードをいっしょにビルドします
  - 次のページを参照して、コピーした etrobo_tr_ex1 用の Makefile の次の箇所を書き換えます
    - ターゲット実行形式ファイル名を、etrobo_tr_ex2 に変更します
    - C++ソースファイルに、新たに追加したクラスの実装ファイル名を追加し、不要なファイル名を外します
  - ビルド手順については、4章を参考にしてください
    - 「app」がビルドできればOKです
- 完成したプログラムを走行体に転送して実行します
  - 転送手順については、4章を参考にしてください
  - 転送できないとき
    - 本体に挿入したSDカードの空きが不足しているかもしれません
      - USBで転送する場所は、本体に挿入したSDカードなのを思い出しましょう
    - これまでの演習で使ったファイルを削除してから転送してみましょう

6-15. ビルドし走行体を動かす（2）

EV3 ET ROBOT CONTEST

テスト

- Makefile.incを修正します
- ビルド手順については、4章を参考にしてください
  - 「app」がビルドできればOKです

```
Makefile.inc

 （途中省略）
APPL_CXXOBJS += ¥
        BalancingWalker.o ¥
        BalancerCpp.o ¥
        Starter.o ¥
        SimpleTimer.o
        ScenarioTracer.o ¥
        Scenario.o ¥
        Scene.o

 （以下省略）
```

LineMonitor.o、
LineTracer.o、
LineTracerWithStarter.o
を対象から外します

追加したクラスの実装ファイル（*.o）を追加します

理解度チェック（6）

- システムの機能を変更する

1. 新たな機能追加要求を実現するために、どの開発工程、どのモデルで何を行ったでしょうか？
   a. まずは
   b. 次に
   c. そして
   d. 　　：
2. 実際に走らせてみたら、シナリオ通りに走行は切り替わるものの、期待した四角い走行軌跡にはなりませんでした
   a. 四角い走行軌跡にならなかった原因はなんでしょうか
   b. どうしたら想定した四角い軌跡通りに走るようになるでしょうか
   c. 原因を取り除いて想定通り走らせるには、どの工程のどのモデルを見直せばよいでしょうか