

2017年11月28日（火）  
RTミドルウェア講習会@人工知能学会



# 第3部：RTシステム構築実習

国立研究開発法人産業技術総合研究所  
ロボットイノベーション研究センター  
ロボットソフトウェアプラットフォーム研究チーム  
高橋 三郎

# 目次

1. 事前確認
2. 実習概要
3. 実習A
4. 実習B
5. 応用課題

# 1. 事前確認

# 資料 (USBメモリで配布)

- スライド
  - 現在見せているスライドと同じもの
- WEBページ
  - 手順を記載したページ
    - チュートリアル(AIツール、第3部、「データ収集・蓄積」の実習) \_ [OpenRTM-aist.htm](#)
    - チュートリアル(AIツール、第3部、「推論結果の検証」の実習) \_ [OpenRTM-aist.htm](#)
- OpenRTM-aistインストーラ(Windows用)
  - OpenRTM-aistのインストールに問題があった場合に使用してください
- Pythonパッケージ (Chainer, OpenCV-Python)
  - この講習で使うPythonパッケージです。事前にインストールされていない方は下記を解凍し、作成されたフォルダに移動後、下記コマンドでインストールして下さい。
    - `python setup.py install`
- EXE
  - RaspberryPiMouseSimulatorComp.exe(シミュレータ)
- sample
  - 本実習で作成するコンポーネントの見本です
    - NameToVelocity
    - ImageDataCollector

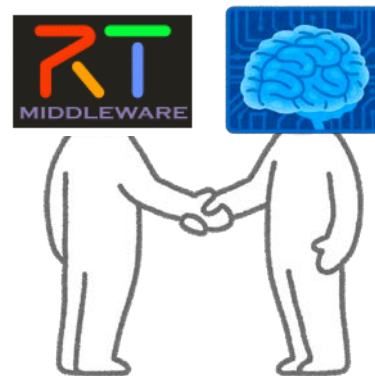
# インストールの確認

- OpenRTM-aist 1.1.2
- Python
- PyYAML
- Doxygen
- Python用エディタ
- Python パッケージ
  - Chainer
  - OpenCV-Python

## 2. 実習概要

# 目標

- RTM (OpenRTM-aist) を利用し、人工知能技術を応用したロボットシステムを構築します
- 深層学習による画像認識を利用した移動ロボット制御システムを作成することで、実際の研究、開発へのアプリケーション応用について学びます



# RTMを使用すべきかのチェックポイント (AIツール観点)

開発するシステムは・・・

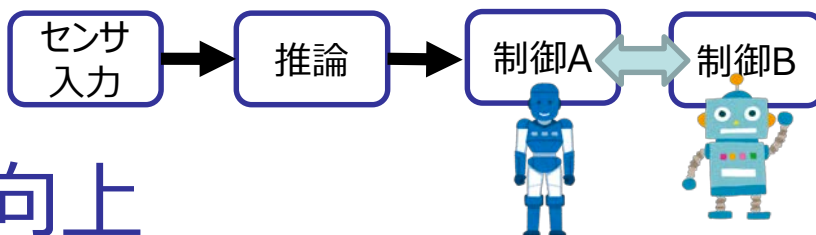
- ロボットを利用するか？
  - － 既存のロボット制御用コンポーネントが利用できる
- 修正を行いながらの繰り返し開発を行うか？
  - － コンポーネント分割により修正を局所化できる
- 第三者が設計情報を再利用するか？
  - － コンポーネント単位で疎結合なため再利用性が高い



# RTMの特長（AIツール観点）

- 設計情報の再利用性向上

- 学習，ロボット制御，データ収集などのロジックを別コンポーネント化することでシステム変更，流用時の修正を局所化



- 分散制御による拡張性向上

- 制御対象ロボットの追加や出力信号の多重化が容易
- 制御周期や通信バッファ方式を変更することで，データ流量や制御タイミングを設定可能



# AIツールとしてRTMを利用する開発の流れ

要求定義

- システム要求定義

- 開発の目的, 要求事項・対象データ明確化

設計・開発

- システム設計・開発

- システム構成の検討, 学習, 推論エンジン開発

☆RTMを活用できる工程

データ収集

- データ収集・蓄積

- 環境構築, 実データの取得

☆RTMを活用できる工程  
実習: 3-B

学習

- 学習 (データ前処理含む)

- データ前処理, 学習処理の実行

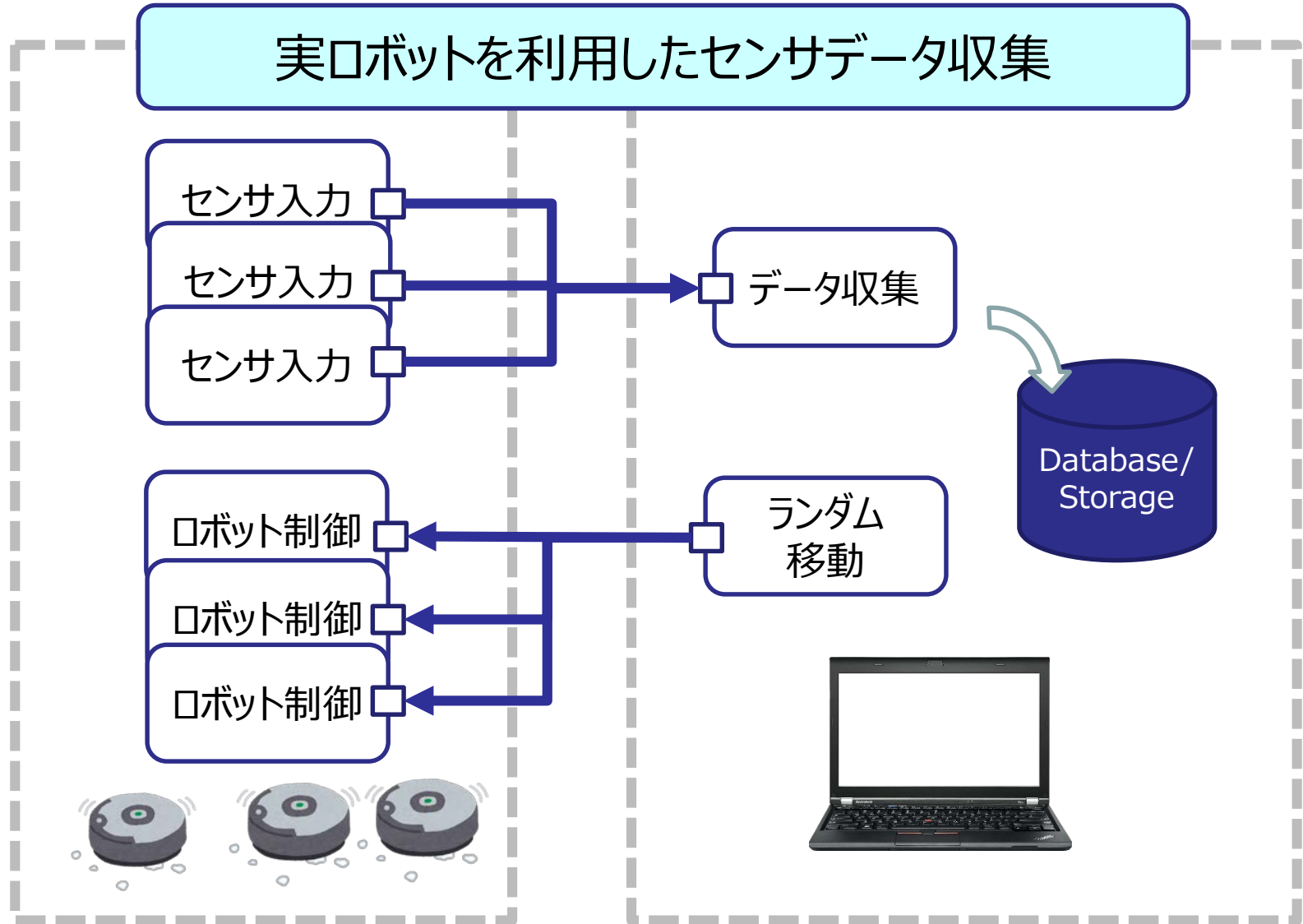
検証

- 推論結果の検証

- 結果妥当性確認, 実環境でのテスト

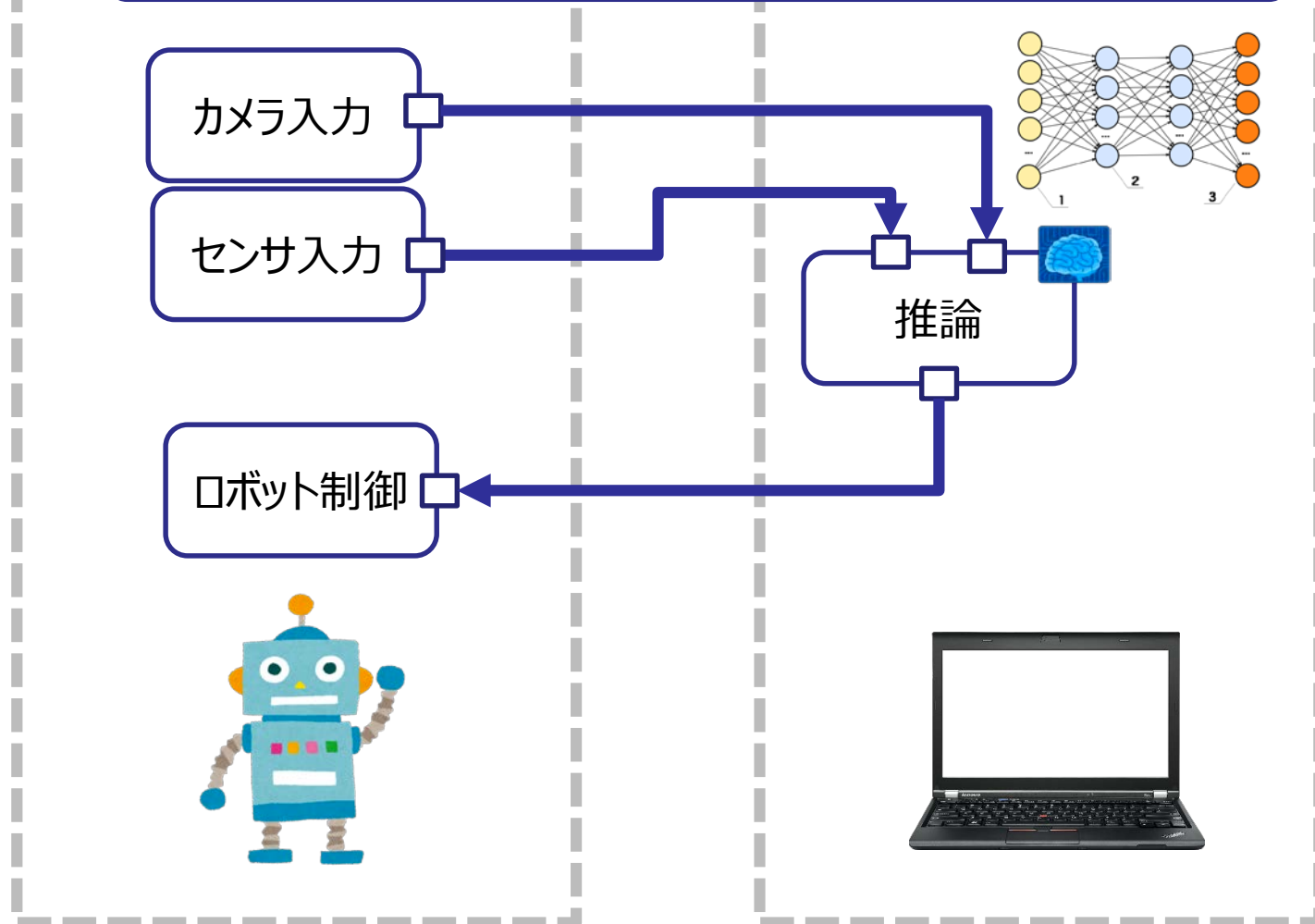
☆RTMを活用できる工程  
実習: 3-A

# 「データ収集・蓄積」での利用例



# 「推論結果の検証」での利用例

## 実ロボットを利用した推論による制御

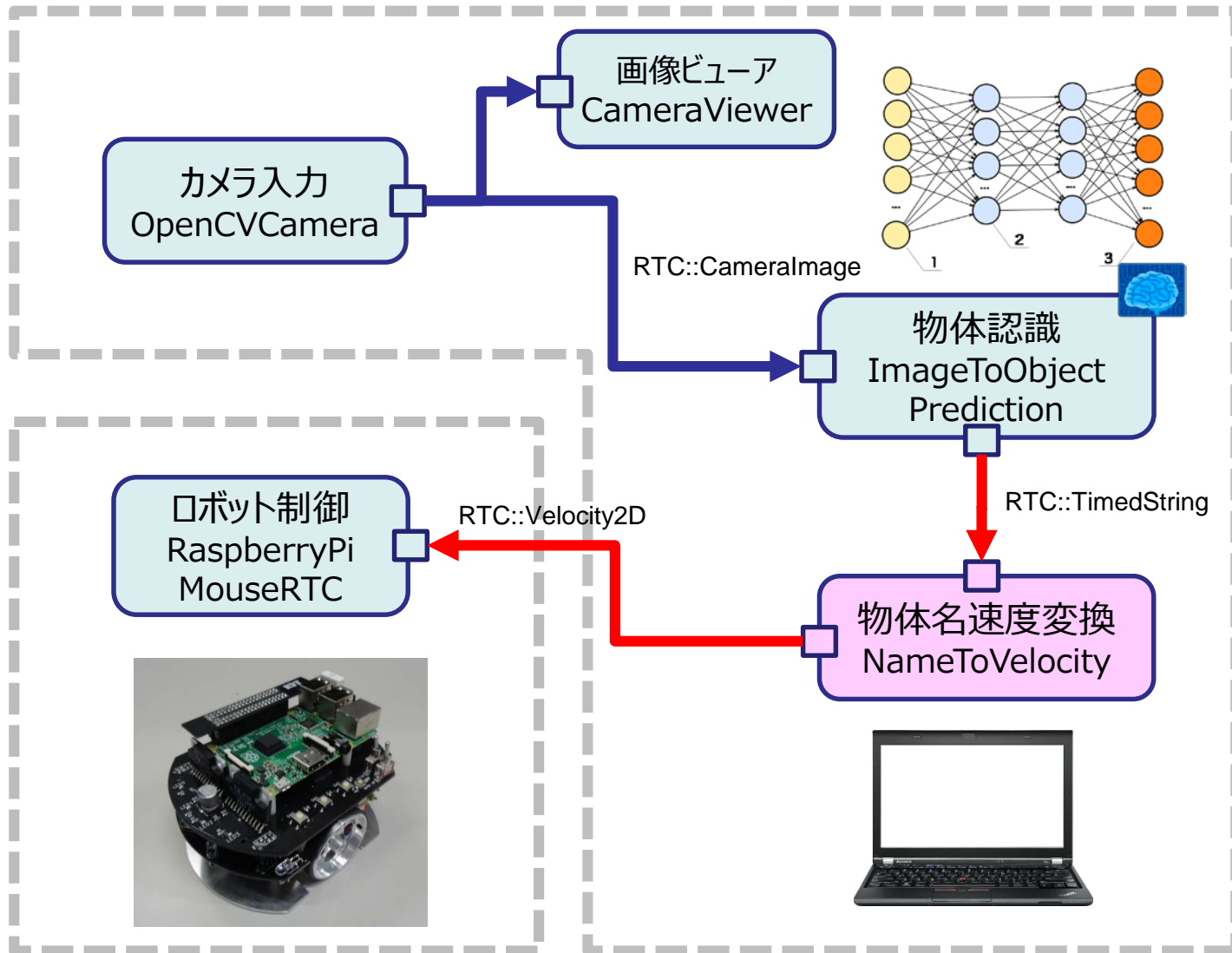




# 3. 実習A

## 「推論結果の検証」

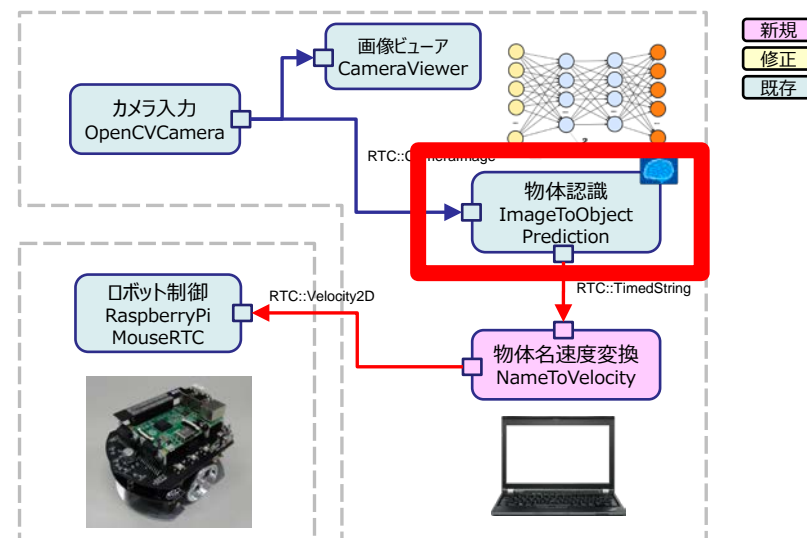
# 【実習3-A】 「推論結果の検証」実習



新規  
修正  
既存

## 【実習3-A】 物体認識コンポーネント (ImageToObjectPrediction) の仕様

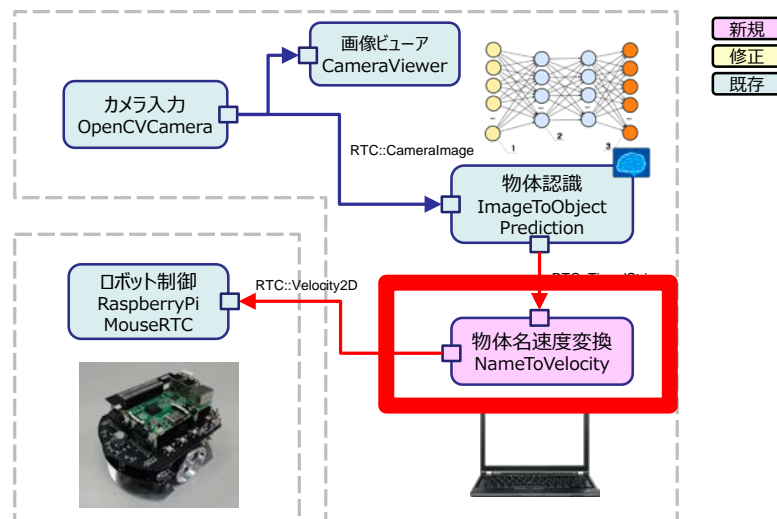
- CNN (GoogLeNet) による物体認識コンポーネント
- 実装は Chainer の ImageNet を利用
- 認識物体名はImageToObjectPrediction/labels.txtに記載
- 入力された画像に対し，学習済みのモデルで物体名を推論
- 本実習では時間の都合上，既に作成済みのものを再利用します





## 【実習3-A】 作成するRTC (NameToVelocity) の仕様

- カメラ画像から認識した物体名が文字列 (TimedString型) として入力されます
- 認識される物体名はImageToObjectPrediction/label.txt記載
  - 認識率が比較的高いのは「bow tie」「hook」「pinwheel」「envelop」でしたので、テスト用に印刷した画像を配布します
- 物体名を速度情報 (TimedVelocity2D型) に読み替えてロボットを制御する
  - 物体名を移動方向に割り当てます  
前進, 右旋回, 左旋回, 後進
  - 割り当てていない物体名を認識した場合は停止します

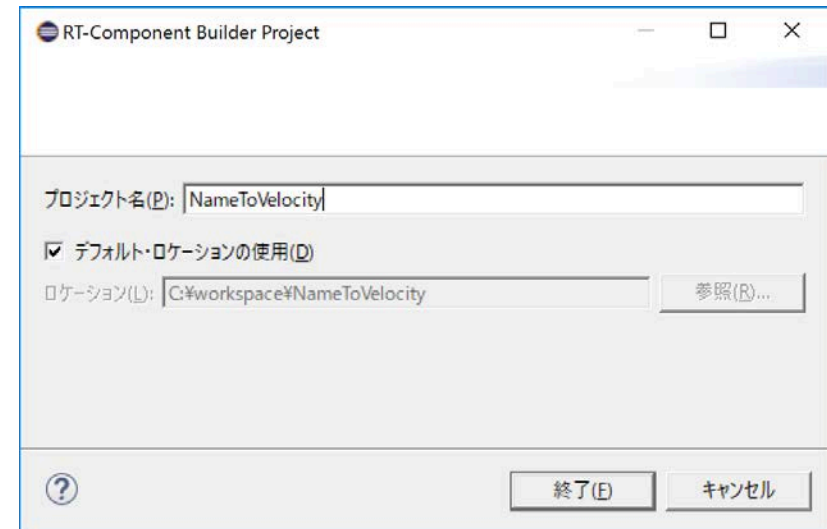
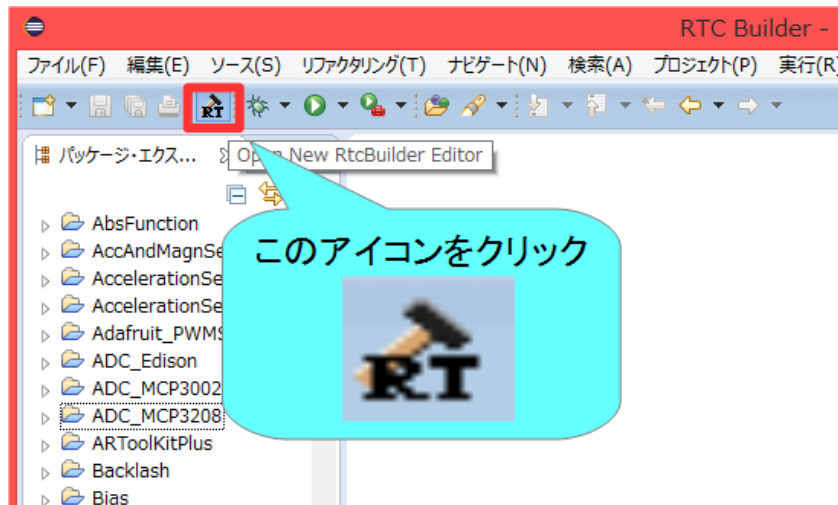


## 【実習3-A】 全体の手順

- RTC Builderによるソースコード等のひな型の作成
- ソースコードの編集
  - NameToVelocity.pyの作成
- RTシステムエディタによるRTシステム作成、動作確認
  - RTシステム作成
    - データポート接続
- Raspberry Piマウスシミュレータとの接続、動作確認
- Raspberry Piマウス実機との接続、動作確認

## 【実習3-A】 プロジェクト作成

- NameToVelocityコンポーネントのスケルトンコードをRTCBuilderで作成する



# 【実習3-A】 基本プロフィールの入力

プロパティ	内容
モジュール名	NameToVelocity
モジュール概要	任意 (Convert to robot velocity from object name)
バージョン	任意 (1.0.0)
ベンダ名	任意
モジュールカテゴリ	任意 (Converter)
コンポーネント型	STATIC
アクティビティ型	PERIODIC
コンポーネントの種類	DataFlow
最大インスタンス数	1
実行型	PeriodicExecutionContext
実行周期	1000.0
概要	任意

# 【実習3-A】 アクティビティの設定

- 以下のアクティビティを有効にする
  - onInitialize
  - **onActivated**
  - **onDeactivated**
  - **onExecute**
- Documentationは適当に書いておいてください
  - 空白でも大丈夫です

▼ アクティビティ

このセクションでは使用するアクションコールバックを指定します。

コンポーネントの初期化と終了処理に関するアクション

onInitialize onFinalize

実行コンテキストの起動と停止に関するアクション

onStartup onShutdown

alive状態でのコンポーネントアクション

onActivated onDeactivated onAborting

onError onReset

Dataflow型コンポーネントのアクション

onExecute onStateUpdate onRateChanged

FSM型コンポーネントのアクション

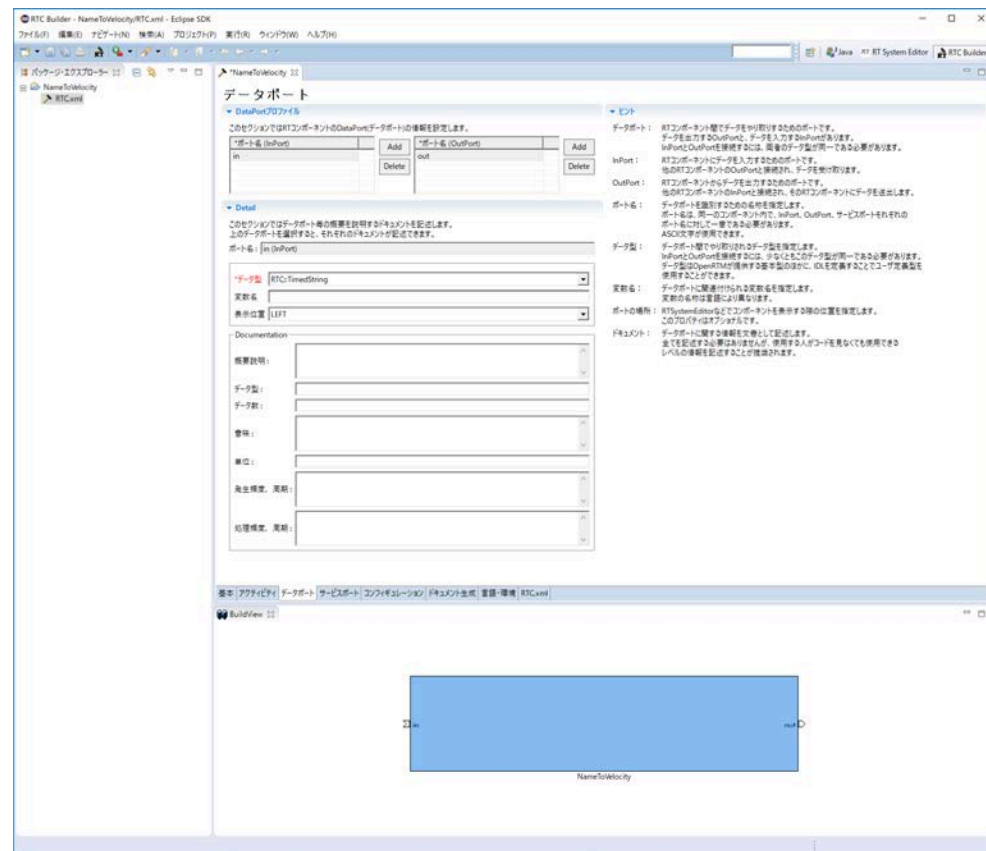
onAction

Mode型コンポーネントのアクション

onModeChanged

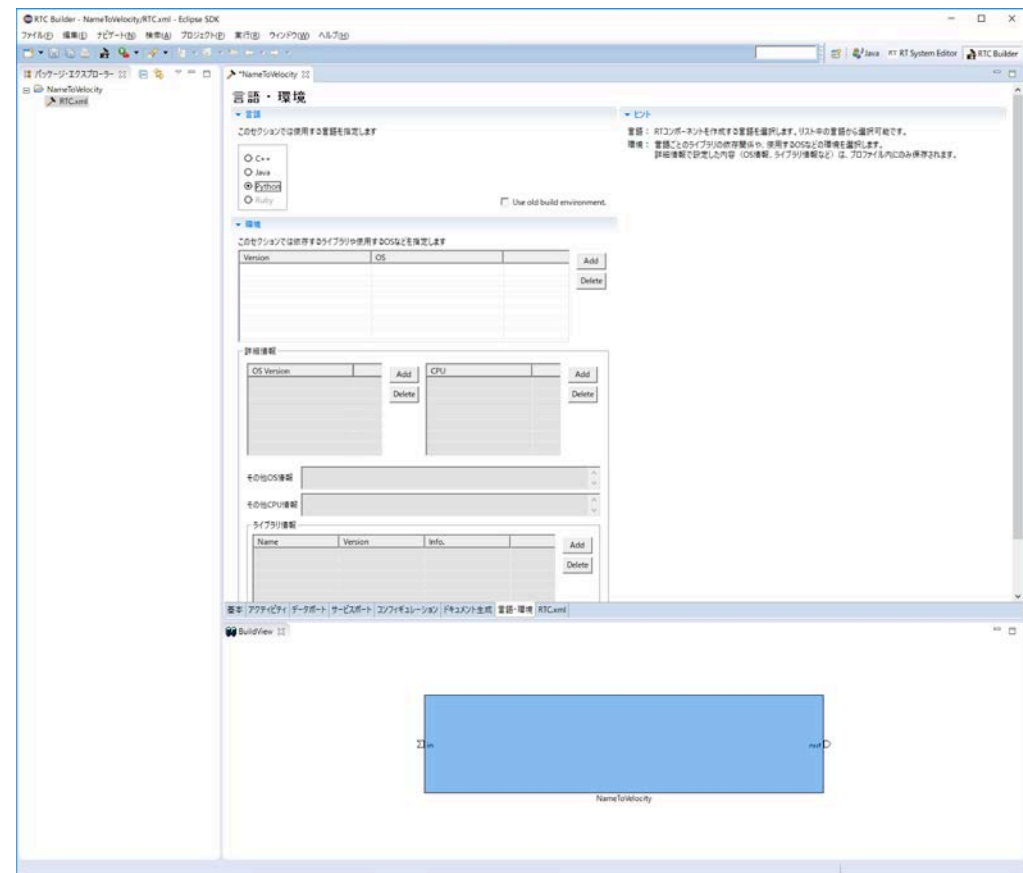
# 【実習3-A】 データポートの設定

- 以下のOutPortを設定する
  - ポート名 : **out**
  - データ型 : **RTC::TimedVelocity2D**
  - 他の項目は任意
- 以下のInPortを設定する
  - ポート名 : **in**
  - データ : **RTC::TimedString**
  - 他の項目は任意



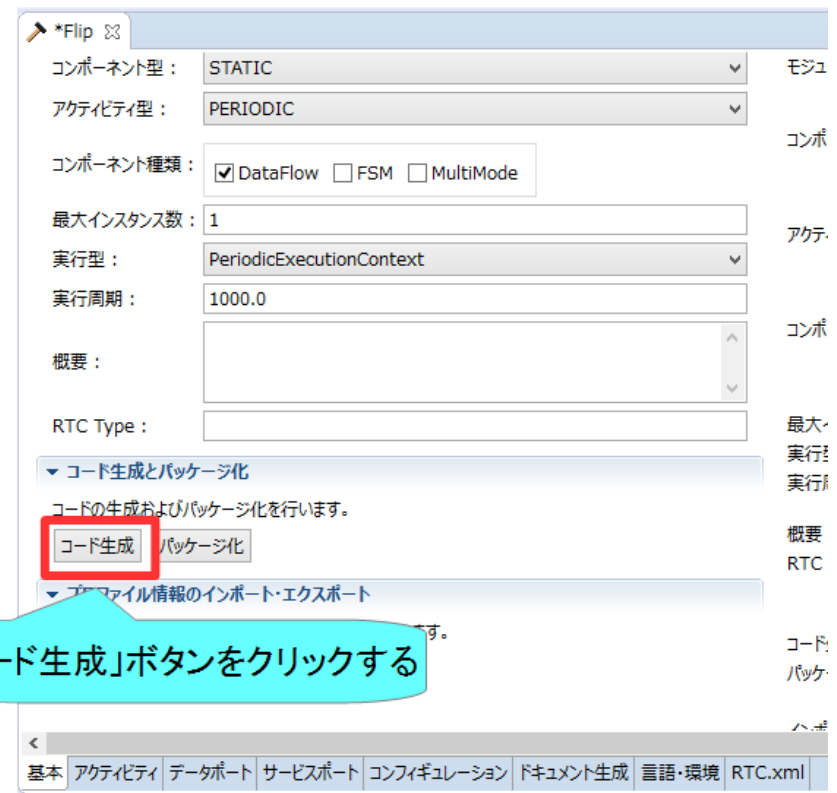
# 【実習3-A】 言語の設定

- 実装する言語，動作環境に関する情報を設定
- Python を選択



# 【実習3-A】スケルトンコードの生成

- 基本タブからコード生成ボタンを押すことでスケルトンコードが生成される
  - Workspace¥NameToVelocity以下に生成
    - Pythonソースファイル(.py)
      - このソースコードにロボットを操作する処理を記述する
    - rtc.conf、NameToVelocity.conf
    - 以下略
  - ファイルが生成できているかを確認してください





## 【実習3-A】 ソースコードの修正①

- NameToVelocity.py をエディタで開き,  
変数初期化部分を下記のように修正します

変数初期化  
(OpenRTM-aist 1.2.0では  
自動生成される予定です)

```
def __init__(self, manager):  
    OpenRTM_aist.DataFlowComponentBase.__init__(self, manager)  
  
    self._d_in = RTC.TimedString(RTC.Time(0,0), "")  
    self._inIn = OpenRTM_aist.InPort("in", self._d_in)  
  
    self._d_out = RTC.TimedVelocity2D(RTC.Time(0,0), RTC.Velocity2D(0.0, 0.0, 0.0))  
    self._outOut = OpenRTM_aist.OutPort("out", self._d_out)
```

## 【実習3-A】 ソースコードの修正②

- NameToVelocity.py をエディタで開き,  
Activate, Deactivate 部分を下記のように修正します
  - 起動, 終了時にロボットを停止するための処理

```
def onActivated(self, ec_id):  
    # ロボットへの出力を初期化しておく  
    self._d_out.data.vy = 0.0  
    self._d_out.data.vx = 0.0  
    self._d_out.data.va = 0.0  
    self._outOut.write()  
  
    return RTC.RTC_OK
```

```
def onDeactivated(self, ec_id):  
    # ロボットを停止する  
    self._d_out.data.vy = 0.0  
    self._d_out.data.vx = 0.0  
    self._d_out.data.va = 0.0  
    self._outOut.write()  
  
    return RTC.RTC_OK
```

起動, 終了時にロボットを停止  
(速度 = 0) するための処理

## 【実習3-A】 ソースコードの修正③

- Execute部分を右記のように修正します

- 認識した文字列に制御速度を割り当てます
- 割り当ててる速度や文字列は自由に変更してみてください

```
def onExecute(self, ec_id):  
    # 入力データが存在するか確認  
    if self._inIn.isNew():  
        # 入力データが存在する場合には、データを別変数に格納  
        data = self._inIn.read()  
        # 入力データの文字列に応じてロボットを操作する  
        if data.data == "bow tie":  
            self._d_out.data.vx = 0.5  
        elif data.data == "hook":  
            self._d_out.data.vx = -0.5  
        elif data.data == "pinwheel":  
            self._d_out.data.va = 0.3  
        elif data.data == "envelope":  
            self._d_out.data.va = -0.3  
        else:  
            self._d_out.data.vx = 0  
            self._d_out.data.vy = 0  
            self._d_out.data.va = 0  
  
        self._outOut.write()  
    return RTC.RTC_OK
```

認識した文字列に対して  
速度を割り当て

## 【実習3-A】 コンポーネントの起動

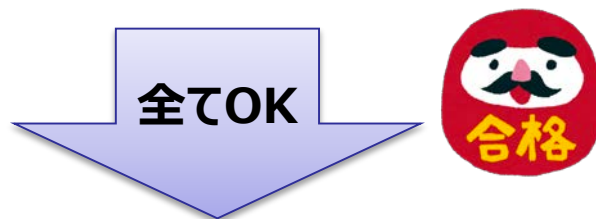
コンポーネント名	起動方法
NameToVelocity	作成したNameToVelocity.pyファイルをダブルクリックもしくはコマンドプロンプトから“python NameToVelocity.py”と入力して起動して下さい。
OpenCVCamera	OpenRTM-aistインストール時に同時にインストールされています。 Windowsの検索（Windowsアイコン押下時のプログラムとファイルの検索など）を用い、OpenCVCameraComp.exeを起動して下さい。
CameraViewer	OpenRTM-aistインストール時に同時にインストールされています。 Windowsの検索（Windowsアイコン押下時のプログラムとファイルの検索など）を用い、CameraViewerComp.exeを起動して下さい。
ImageToObjectPrediction	USBメモリで配布されたsampleフォルダにある ImageToObjectPrediction.pyをダブルクリックして起動して下さい。
RaspberryPiMouseSimulator	USBメモリで配布されたEXEフォルダにある RaspberryPiMouseSimulatorComp.exeをダブルクリックして起動して下さい。

- RTSystemEditor  
を起動し、  
右記のようにポートを  
接続します
- 接続後、コンポーネ  
ントをActivateしま  
す



## 【実習3-A】 動作確認

- 起動したコンポーネントが全てActive状態（緑色）
- カメラ画像が表示される
- カメラ画像が認識された際にImageToObjectPredictionコンポーネントを起動したプロンプト上に"Recognized Object: xxx"と表示される
- コード上に割り当てた物体を認識すると、シミュレータ上のロボットが期待通り動作する



### 実機での動作確認

※時間がある方は、下記にも挑戦ください

- 身の回りのより認識しやすい物体に置き換えて動作
- 制御方法（条件分岐）をより複雑化してなめらかな動作

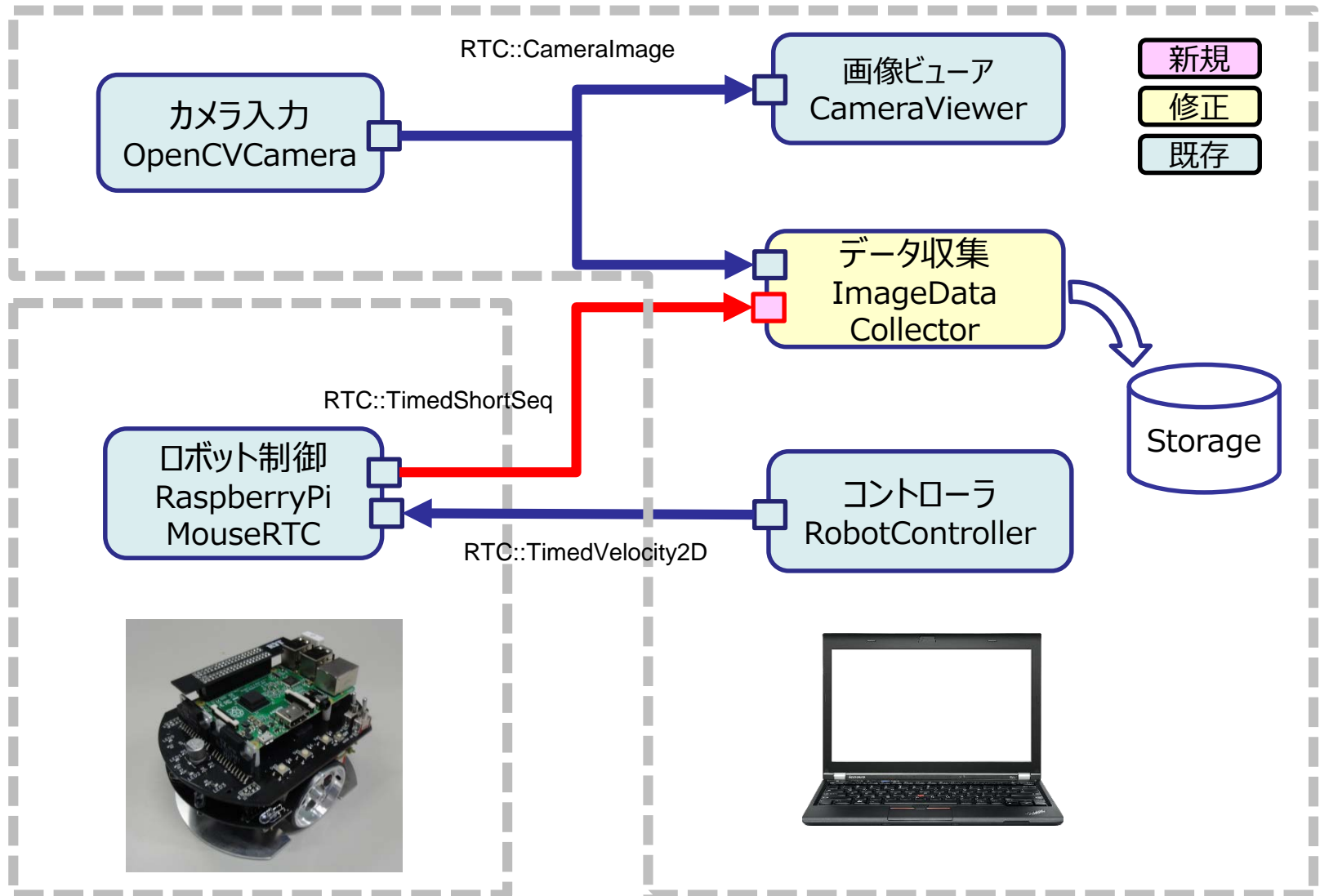


## 4. 実習B

# 「データ収集・蓄積の検証」

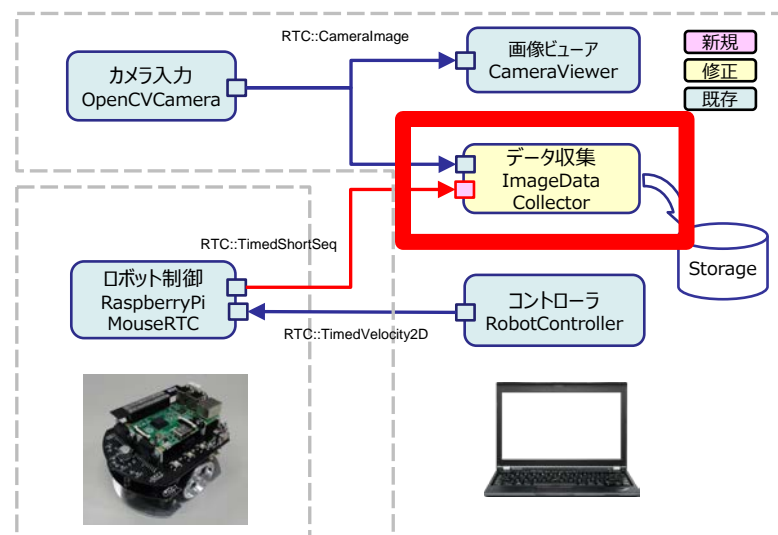


# 【実習3-B】「データ収集・蓄積」実習



## 【実習3-B】 修正するRTC(ImageDataCollector)の仕様

- カメラ画像（CameraImage型）が入力され，入力されたカメラ画像をPNGファイルとして保存します
- ロボットの赤外線センサ情報（TimedShortSeq型）が入力され，入力されたセンサ情報をCSVファイルとして保存します
- 上記処理を100ms毎に行います

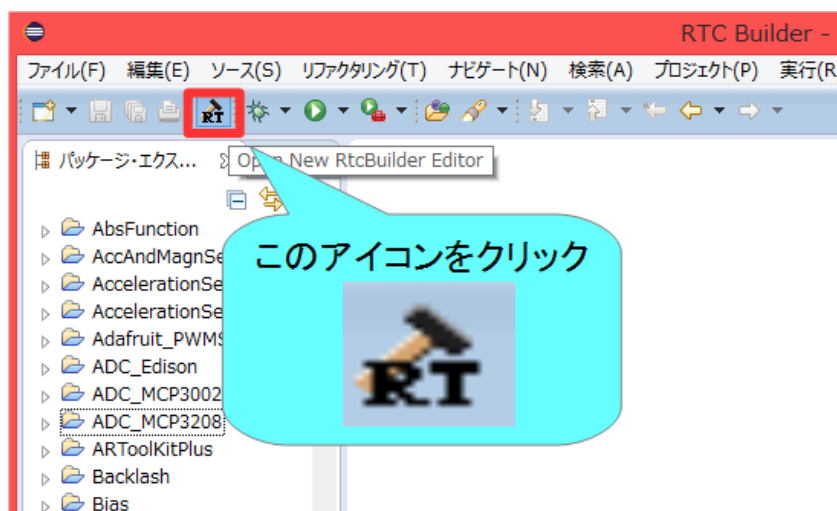


## 【実習3-B】 全体の手順

- RTC Builderによるソースコード等のひな型の作成
  - 既存のプロファイルを再利用
- ソースコードの編集
  - ImageDataCollector.pyの作成
- RTシステムエディタによるRTシステム作成、動作確認
  - RTシステム作成
    - データポート接続
- Raspberry Piマウスシミュレータとの接続、動作確認
- Raspberry Piマウス実機との接続、動作確認

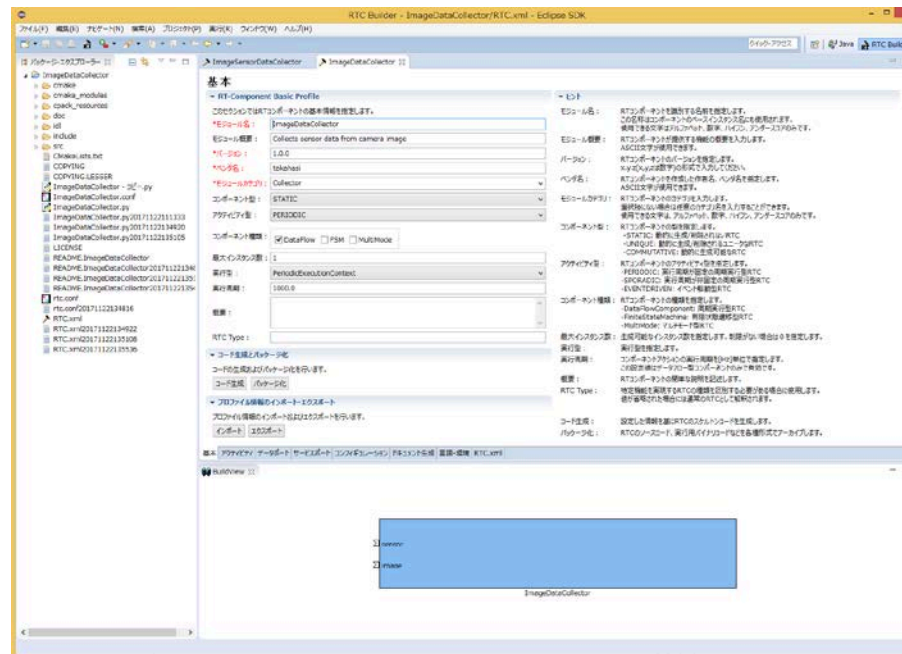
## 【実習3-B】 プロジェクト作成

- ImageDataCollectorコンポーネントのスケルトンコードをRTCBuilderで作成する



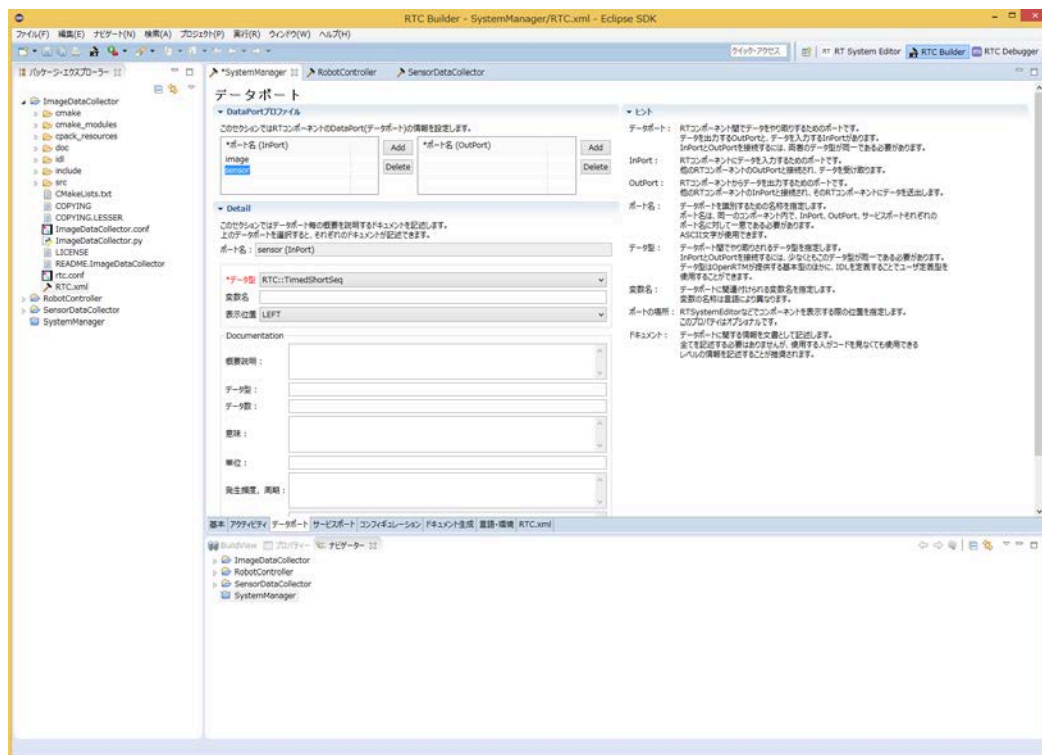
# 【実習3-B】 基本プロファイルの入力

- USBメモリ sampleフォルダ内のImageDataCollectorコンポーネントのRTC.xmlを読み込む
  - RTC Builder には既存コンポーネントのプロファイルを読み込む機能があります
  - 「基本」タブにある「プロファイル情報のインポート・エクスポート」の「インポート」ボタンをクリックし、改造元のImageDataCollectorにあるRTC.xmlを選択



# 【実習3-B】 データポートの設定

- 以下のOutPortを追加する
  - ポート名 : **sensor**
  - データ型 : **RTC::TimedShortSeq**
  - 他の項目は任意
- 以下のOutPortはそのまま維持
  - ポート名 : **image**
  - データ : **RTC::CameraImage**
  - 他の項目は任意



# 【実習3-B】 スケルトンコードの生成

- 実行周期が10になっていることを確認
  - 10Hz=100msごとに動作
- 基本タブからコード生成ボタンを押すことでスケルトンコードが生成される
  - Workspace¥ImageDataCollector以下に生成
    - Pythonソースファイル(.py)
      - このソースコードにロボットを操作する処理を記述する
    - rtc.conf、ImageDataCollector.conf
    - 以下略
  - ファイルが生成できているかを確認

\*Flip

コンポーネント型: STATIC

アクティビティ型: PERIODIC

コンポーネント種類: ☒ DataFlow ☐ FSM ☐ MultiMode

最大インスタンス数: 1

実行型: PeriodicExecutionContext

実行周期: 10

概要:

RTC Type:

▼ コード生成とパッケージ化

コードの生成およびパッケージ化を行います。

コード生成 パッケージ化

▼ プロファイル情報のインポート・エクスポート

基本 アクティビティ データポート サービスポート コンフィギュレーション ドキュメント生成 言語・環境 RTC.xml

## 【実習3-B】 ソースコードの修正①

- ImageDataCollector.py をエディタで開き,  
変数初期化部分を下記のように修正します

```
# Import RTM module
import RTC
import OpenRTM_aist
import cv2
import numpy as np
from datetime import datetime as dt
```

画像変換用 + 日付情報取得  
用にimportを追加

変数初期化  
(OpenRTM-aist 1.2.0では  
自動生成される予定です)

```
def __init__(self, manager):
    OpenRTM_aist.DataFlowComponentBase.__init__(self, manager)

    self._d_image = RTC.CameraImage(RTC.Time(0, 0), 0, 0, 0, [], 0, [])
    self._imageIn = OpenRTM_aist.InPort("image", self._d_image)

    self._d_sensor = RTC.TimedShortSeq(RTC.Time(0, 0), [])
    self._sensorIn = OpenRTM_aist.InPort("sensor", self._d_sensor)
```



## 【実習3-B】 ソースコードの修正②

- ImageDataCollector.py をエディタで開き，下記のように修正します

```
def onActivated(self, ec_id):  
    # 変数の初期化  
    self._count = 0  
    t = dt.now()  
    # 画像用出力ディレクトリ作成  
    self._image_dir = "image_" + t.strftime('%Y%m%d')  
    if not os.path.exists(self._image_dir):  
        os.mkdir(self._image_dir)  
    # センサ用出力ディレクトリ作成  
    self._sensor_dir = "sensor_" + t.strftime('%Y%m%d')  
    if not os.path.exists(self._sensor_dir):  
        os.mkdir(self._sensor_dir)  
    self._f = open(self._sensor_dir + "/sensor.csv", 'w')  
    return RTC.RTC_OK
```

出力先として日付を名前にしたディレクトリを作成

センサ出力ファイルを開いておく  
(書き込み毎オープン・クローズすると遅いため)

```
def onDeactivated(self, ec_id):  
    self._count = 0  
    self._f.close()  
    return RTC.RTC_OK
```

センサ出力ファイルを閉じる

## 【実習3-B】 ソースコードの修正③

- ImageDataCollector.py をエディタで開き，下記のように修正します

```
def onExecute(self, ec_id):  
    if self._imageIn.isNew():  
        # 画像イメージの変換  
        data = self._imageIn.read()  
        frame = np.frombuffer(data.pixels, dtype=np.uint8)  
        frame = frame.reshape(data.height, data.width, 3)  
        # 画像イメージの保存  
        cv2.imwrite(self._image_dir + "/" + str(self._count) + ".png", frame)  
        self._count += 1  
  
    if self._sensorIn.isNew():  
        # センサデータの保存  
        data = self._sensorIn.read()  
        self._f.write(",".join(map(str, data.data)) + "¥n")  
  
    return RTC.RTC_OK
```

画像をファイル形式に  
整形して保存

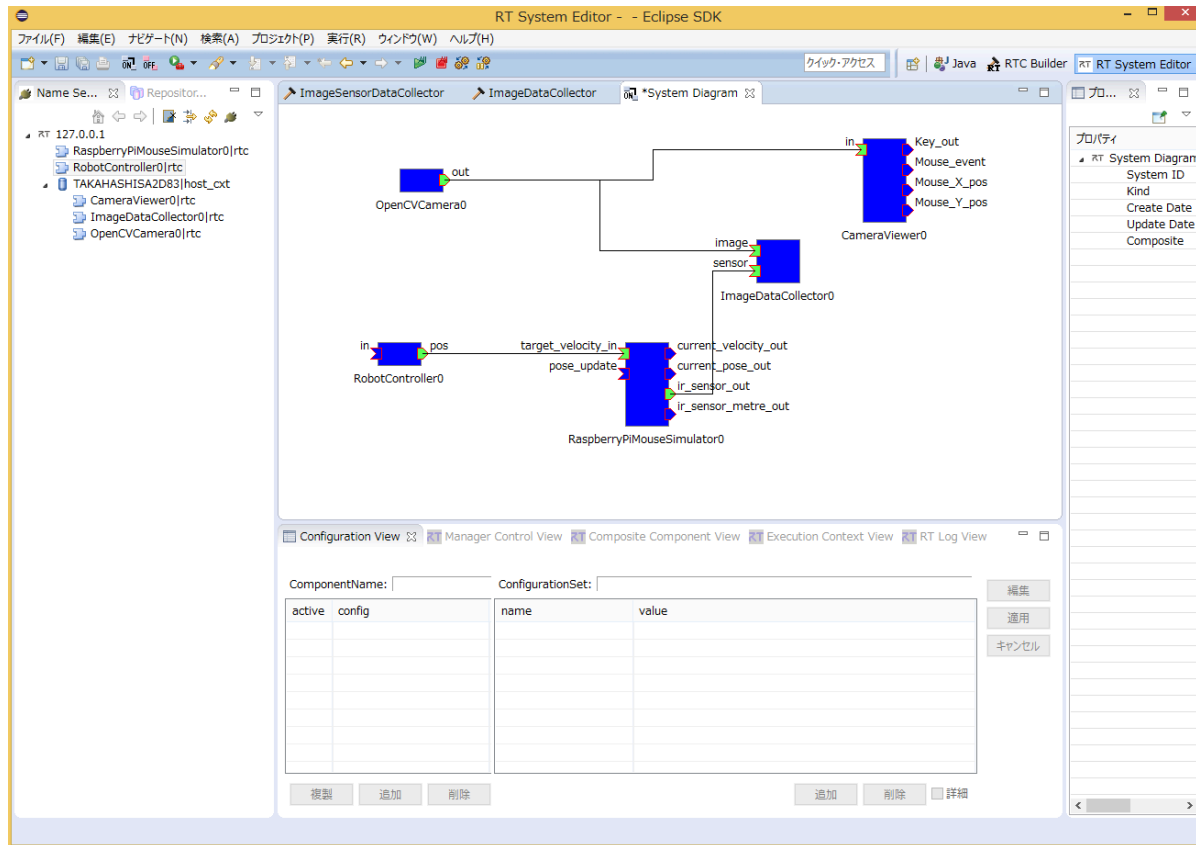
センサ出力ファイルに  
CSV形式で追記

## 【実習3-B】 コンポーネントの起動

コンポーネント名	起動方法
ImageDataCollector	作成したImageDataCollector.pyファイルをダブルクリックもしくはコマンドプロンプトから“python ImageDataCollector.py”と入力して起動して下さい。
OpenCVCamera	OpenRTM-aistインストール時に同時にインストールされています。 Windowsの検索（Windowsアイコン押下時のプログラムとファイルの検索など）を用い、OpenCVCameraComp.exeを起動して下さい。
CameraViewer	OpenRTM-aistインストール時に同時にインストールされています。 Windowsの検索（Windowsアイコン押下時のプログラムとファイルの検索など）を用い、CameraViewerComp.exeを起動して下さい。
RobotController	第二部で作成したコンポーネントです。 RobotController.pyをダブルクリックして起動して下さい。
RaspberryPiMouseSimulator	USBメモリで配布されたEXEフォルダにある RaspberryPiMouseSimulatorComp.exeをダブルクリックして起動して下さい。

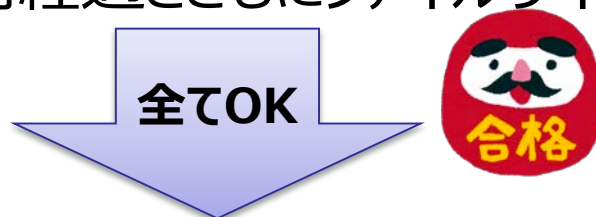
## 【実習3-B】 データポートの接続

- RTSystemEditorを起動し，下記のようにポートを接続します
- 接続後，コンポーネントをActivateします



## 【実習3-B】 動作確認

- 起動したコンポーネントが全てActive状態（緑色）
- カメラ画像が表示される
- RobotController のスライダーでロボットが制御できる
- ImageDataCollectorを起動したフォルダ以下にカメラ画像ファイルが生成され，時間経過とともにファイル数が増える
- ImageDataCollectorを起動したフォルダ以下にセンサデータファイルが生成され，時間経過とともにファイルサイズが増える



### 実機での動作確認

※時間がある方は，下記にも挑戦ください

- OpenCVCameraのconfigurationを変更し  
カメラ画像のサイズを大きく or 小さくしてみる



# 5. 応用課題

## 【応用課題①】 RaspberryPiMouseのカメラを使う

- RaspberryPiMouse に USB カメラを接続し，SSH でログインして，OpenCVCameraコンポーネントを起動します
- 実習A or B と同じようにコンポーネントを接続して動作させます



## 【応用課題②】 認識成功時の画像を保存する

- ImageDataCollector と ImageToObjectPredictionのコードをマージして、認識に成功した場合の画像を保存するコンポーネントを作成する