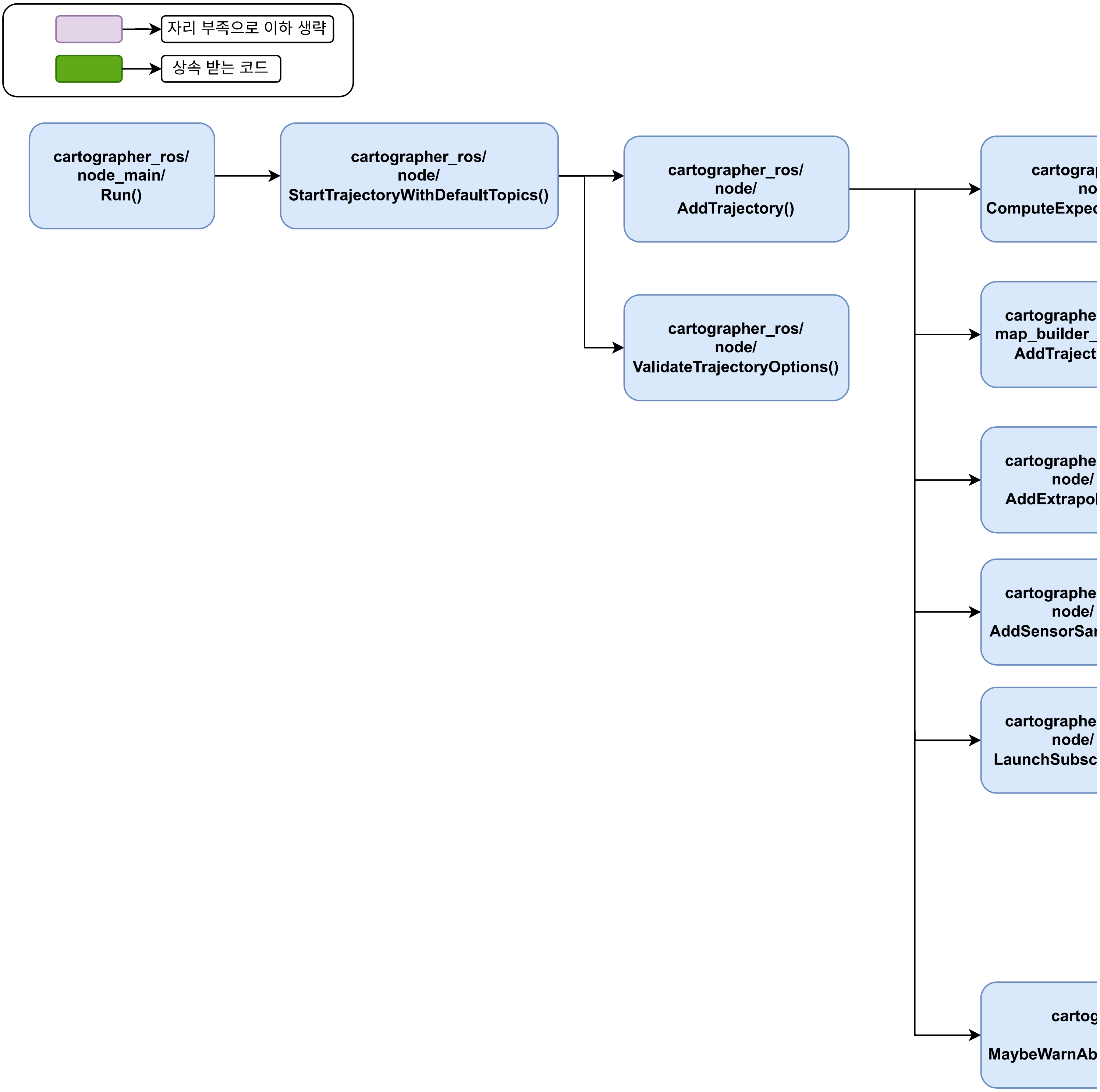


자리 부족으로 이하 생략

상속 받는 코드

cartographer_ros/
node_main/
Run()

cartographer_ros/
node/
StartTrajectoryWithDefaultTopics()

cartographer_ros/
node/
AddTrajectory()

cartographer_ros/
node/
ValidateTrajectoryOptions()

cartogra
no
ComputeExpec

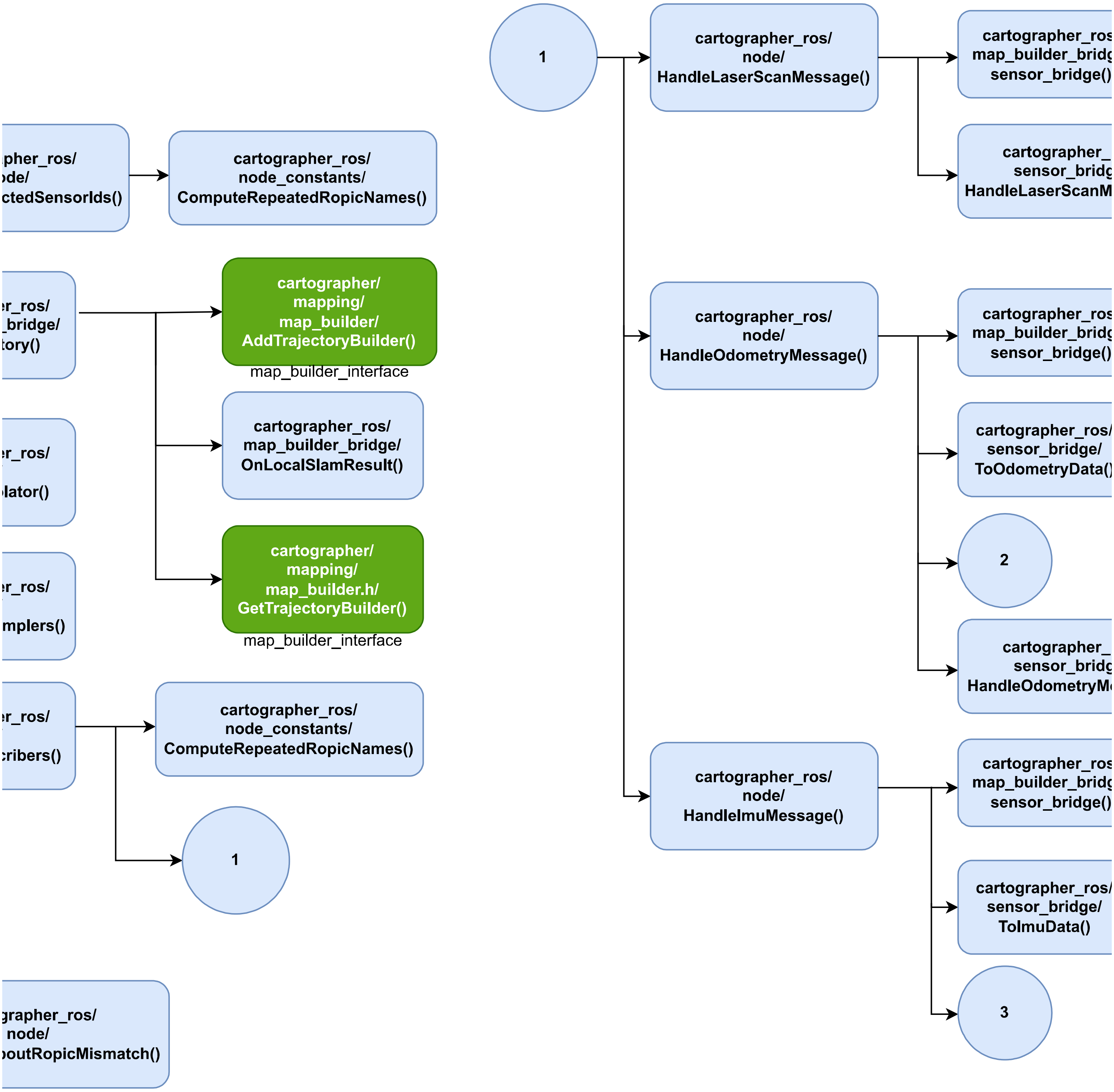
cartographe
map_builder_
AddTraject

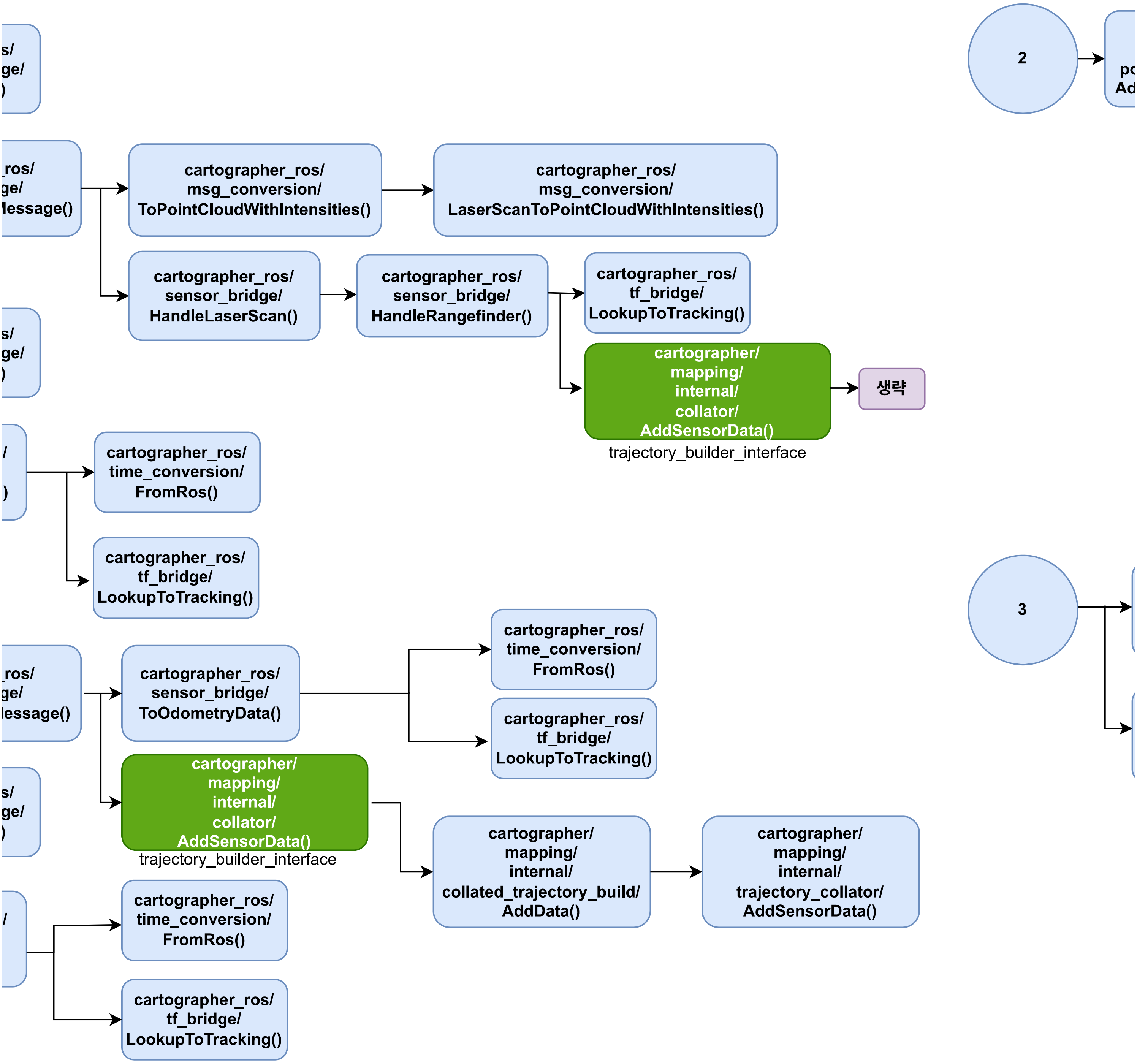
cartographe
node/
AddExtrapol

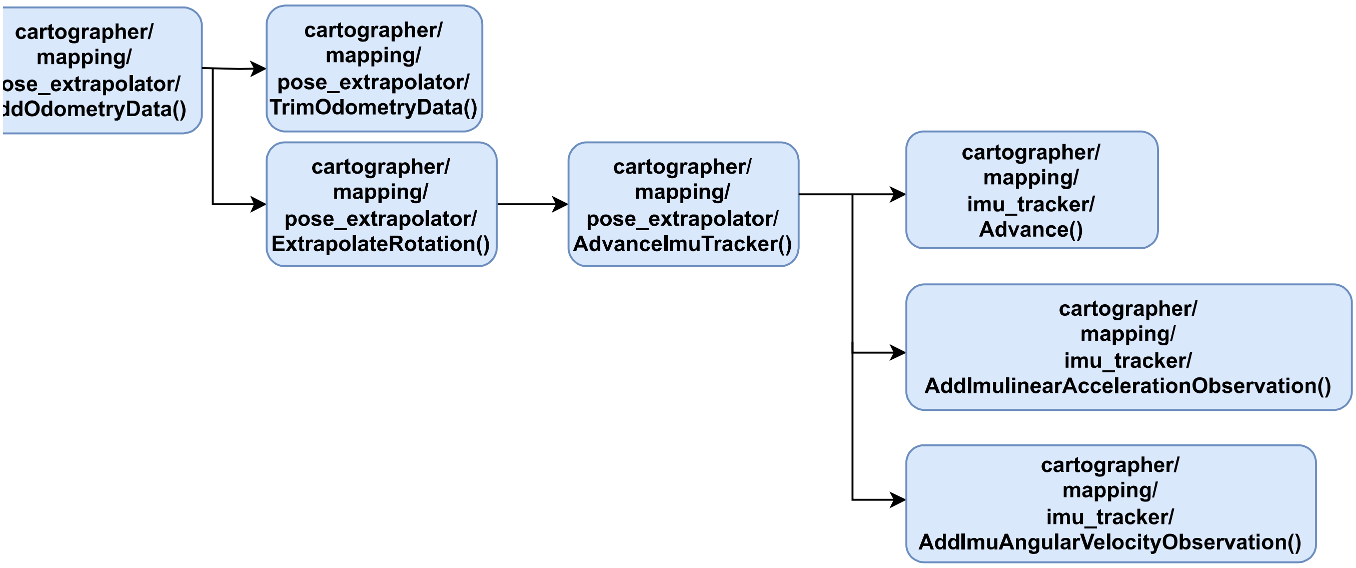
cartographe
node/
AddSensorSa

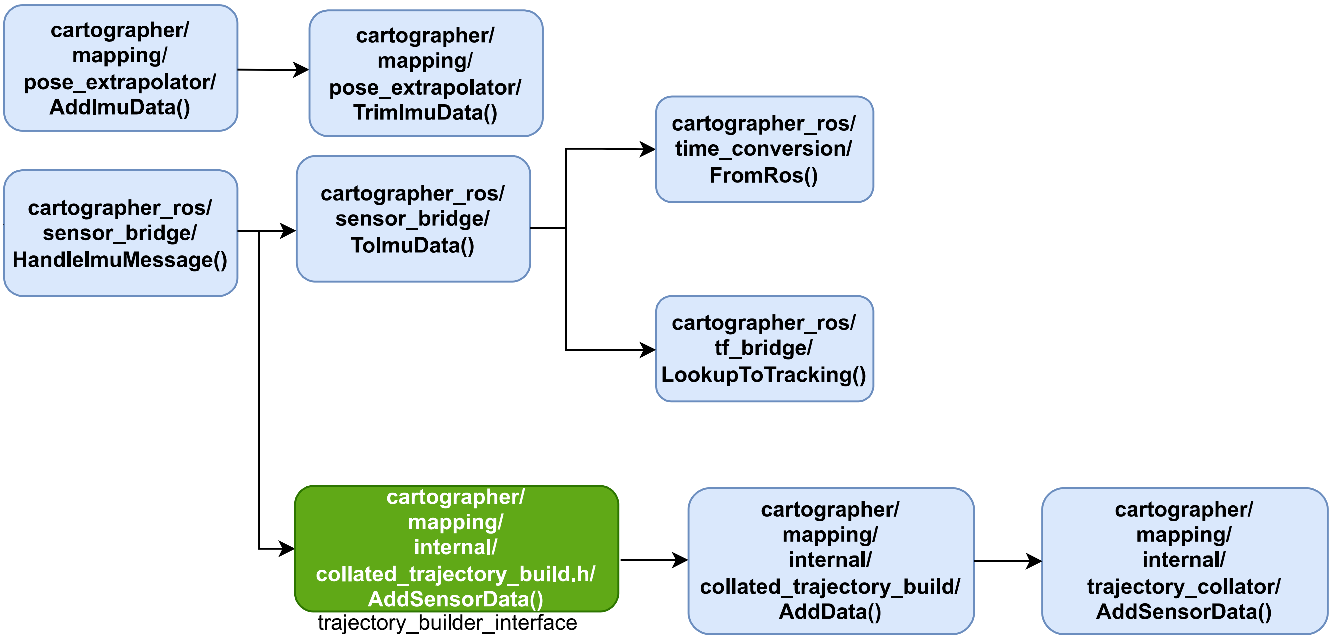
cartographe
node/
LaunchSubsc

cartog
MaybeWarnAb


pher_ros/
de/
ctedSensorIds()
cartographer_ros/
node_constants/
ComputeRepeatedRopicNames()
er_ros/
_bridge/
tory()
cartographer/
mapping/
map_builder/
AddTrajectoryBuilder()
map_builder_interface
er_ros/
lator()
cartographer_ros/
map_builder_bridge/
OnLocalSlamResult()
er_ros/
mplers()
cartographer/
mapping/
map_builder.h/
GetTrajectoryBuilder()
map_builder_interface
er_ros/
cribers()
cartographer_ros/
node_constants/
ComputeRepeatedRopicNames()
1
grapher_ros/
node/
boutRopicMismatch()
1
cartographer_ros/
node/
HandleLaserScanMessage()
cartographer_ros
map_builder_bridg
sensor_bridge()
cartographer_
sensor_bridg
HandleLaserScanM
cartographer_ros/
node/
HandleOdometryMessage()
cartographer_ros
map_builder_bridg
sensor_bridge()
cartographer_ros/
sensor_bridge/
ToOdometryData()
2
cartographer_
sensor_bridg
HandleOdometryM
cartographer_ros/
node/
HandleImuMessage()
cartographer_ros
map_builder_bridg
sensor_bridge()
cartographer_ros/
sensor_bridge/
ToImuData()
3


s/
ge/
)
ros/
ge/
lessage()
cartographer_ros/
msg_conversion/
ToPointCloudWithIntensities()
cartographer_ros/
msg_conversion/
LaserScanToPointCloudWithIntensities()
cartographer_ros/
sensor_bridge/
HandleLaserScan()
cartographer_ros/
sensor_bridge/
HandleRangefinder()
cartographer_ros/
tf_bridge/
LookupToTracking()
cartographer/
mapping/
internal/
collator/
AddSensorData()
trajectory_builder_interface
생략
2
po
Ad
s/
ge/
)
/
)
cartographer_ros/
time_conversion/
FromRos()
cartographer_ros/
tf_bridge/
LookupToTracking()
3
ros/
ge/
lessage()
cartographer_ros/
sensor_bridge/
ToOdometryData()
cartographer_ros/
time_conversion/
FromRos()
cartographer_ros/
tf_bridge/
LookupToTracking()
s/
ge/
)
cartographer/
mapping/
internal/
collator/
AddSensorData()
trajectory_builder_interface
cartographer/
mapping/
internal/
collated_trajectory_build/
AddData()
cartographer/
mapping/
internal/
trajectory_collator/
AddSensorData()
/
cartographer_ros/
time_conversion/
FromRos()
cartographer_ros/
tf_bridge/
LookupToTracking()


cartographer/ mapping/ ose_extrapolator/ ddOdometryData()
cartographer/ mapping/ pose_extrapolator/ TrimOdometryData()
cartographer/ mapping/ pose_extrapolator/ ExtrapolateRotation()
cartographer/ mapping/ pose_extrapolator/ AdvanceImuTracker()
cartographer/ mapping/ imu_tracker/ Advance()
cartographer/ mapping/ imu_tracker/ AddImulinearAccelerationObservation()
cartographer/ mapping/ imu_tracker/ AddImuAngularVelocityObservation()

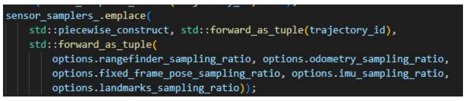

cartographer/ mapping/ pose_extrapolator/ AddImuData()
cartographer/ mapping/ pose_extrapolator/ TrimImuData()
cartographer_ros/ sensor_bridge/ HandleImuMessage()
cartographer_ros/ sensor_bridge/ ToImuData()
cartographer_ros/ time_conversion/ FromRos()
cartographer_ros/ tf_bridge/ LookupToTracking()
cartographer/ mapping/ internal/ collated_trajectory_build.h/ AddSensorData()
trajectory_builder_interface
cartographer/ mapping/ internal/ collated_trajectory_build/ AddData()
cartographer/ mapping/ internal/ trajectory_collator/ AddSensorData()

→ 포인터에 값을 저장하는 경우

→ 포인터로 부터 값을 가져오는 경우

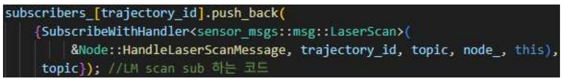
<node.cpp>

cartographer_ros/node/AddSensorSamplers() → sensor_samplers_



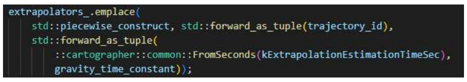
```
sensor_samplers_.emplace(
    std::piecewise_construct, std::forward_as_tuple(trajectory_id),
    std::forward_as_tuple(
        options.rangefinder_sampling_ratio, options.odometry_sampling_ratio,
        options.fixed_frame_pose_sampling_ratio, options.imu_sampling_ratio,
        options.landmarks_sampling_ratio));
```

cartographer_ros/node/LaunchSubscriber() → subscriber_



```
subscribers_[trajectory_id].push_back(
    {SubscribeWithHandler<sensor_msgs::msg::LaserScan>(
         &Node::HandleLaserScanMessage, trajectory_id, topic, node_, this),
     topic}); //LM scan sub 하는 코드
```

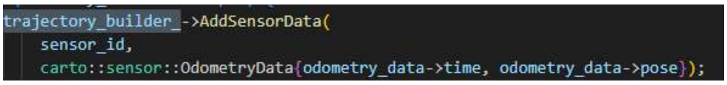
cartographer_ros/node/AddExtrapolator() → extrapolators_



```
extrapolators_.emplace(
    std::piecewise_construct, std::forward_as_tuple(trajectory_id),
    std::forward_as_tuple(
        ::cartographer::common::FromSeconds(kExtrapolationEstimationTimeSec),
        gravity_time_constant));
```

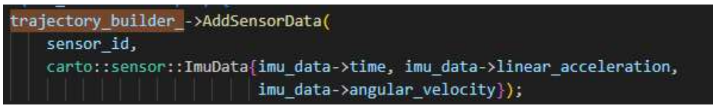
<sensor_bridge.cpp>

cartographer_ros/sensor_bridge/HandleOdometryMessage() → imu_tracker_



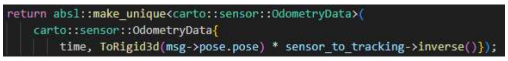
```
trajectory_builder_->AddSensorData(
    sensor_id,
    carto::sensor::OdometryData{odometry_data->time, odometry_data->pose});
```

cartographer_ros/sensor_bridge/HandleImuMessage() → trajectory_builder_



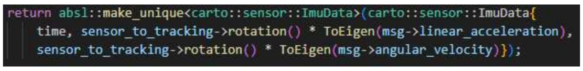
```
trajectory_builder_->AddSensorData(
    sensor_id,
    carto::sensor::ImuData{imu_data->time, imu_data->linear_acceleration,
                           imu_data->angular_velocity});
```

cartographer_ros/sensor_bridge/ToOdometryData() → extrapolators_



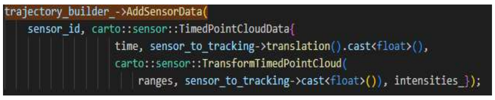
```
return absl::make_unique<carto::sensor::OdometryData>(
    carto::sensor::OdometryData{
        time, ToRigid3d(msg->pose.pose) * sensor_to_tracking->inverse()});
```

cartographer_ros/sensor_bridge/ToImuData() → extrapolators_



```
return absl::make_unique<carto::sensor::ImuData>(carto::sensor::ImuData{
    time, sensor_to_tracking->rotation() * ToEigen(msg->linear_acceleration),
    sensor_to_tracking->rotation() * ToEigen(msg->angular_velocity)});
```

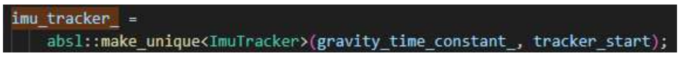
cartographer_ros/sensor_bridge/HandleRangefinder() → trajectory_builder_



```
trajectory_builder_->AddSensorData(
    sensor_id, carto::sensor::TimedPointCloudData{
                   time, sensor_to_tracking->translation().cast<float>(),
                   carto::sensor::TransformTimedPointCloud(
                       ranges, sensor_to_tracking->cast<float>()), intensities_});
```

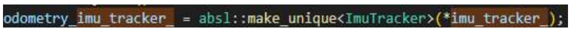
<pose_extrapolator.cc>

cartographer/mapping/pose_extrapolator/AddPose() → imu_tracker_



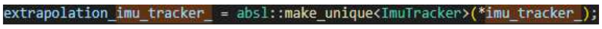
```
imu_tracker_ =
    absl::make_unique<ImuTracker>(gravity_time_constant_, tracker_start);
```

cartographer/mapping/pose_extrapolator/AddPose() → odometry_imu_tracker



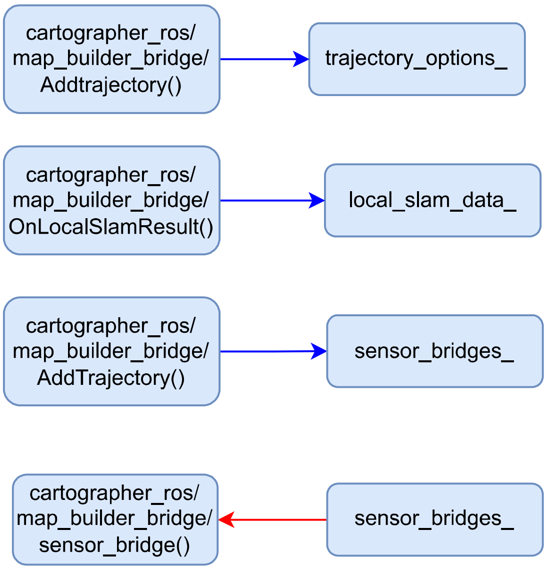
```
odometry_imu_tracker_ = absl::make_unique<ImuTracker>(*imu_tracker_);
```

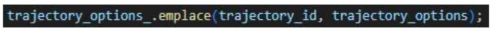
cartographer/mapping/pose_extrapolator/AddPose() → extrapolation_imu_tracker



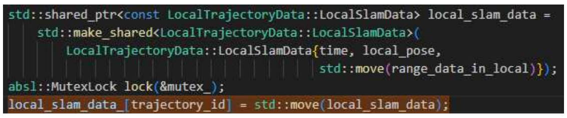
```
extrapolation_imu_tracker_ = absl::make_unique<ImuTracker>(*imu_tracker_);
```

<map_builder_bridge.cpp>

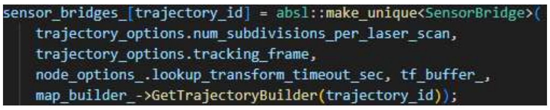




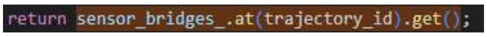
```
trajectory_options_.emplace(trajectory_id, trajectory_options);
```



```
std::shared_ptr<const LocalTrajectoryData::LocalSlamData> local_slam_data =
    std::make_shared<LocalTrajectoryData::LocalSlamData>(
        LocalTrajectoryData::LocalSlamData{time, local_pose,
                                           std::move(range_data_in_local)});
absl::MutexLock lock(&mutex_);
local_slam_data_[trajectory_id] = std::move(local_slam_data);
```

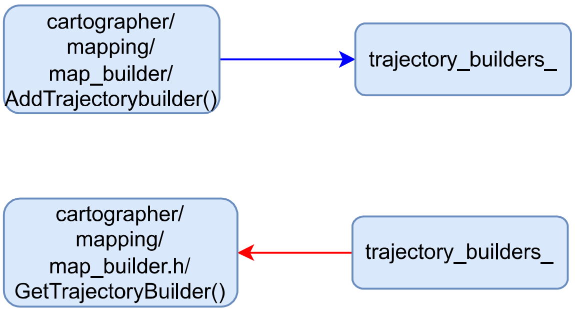


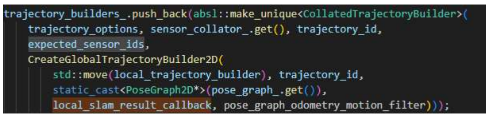
```
sensor_bridges_[trajectory_id] = absl::make_unique<SensorBridge>(
    trajectory_options.num_subdivisions_per_laser_scan,
    trajectory_options.tracking_frame,
    node_options_.lookup_transform_timeout_sec, tf_buffer_,
    map_builder_->GetTrajectoryBuilder(trajectory_id));
```



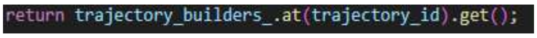
```
return sensor_bridges_.at(trajectory_id).get();
```

<map_builder.cc, .h>

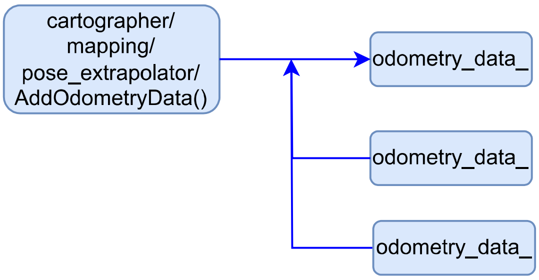


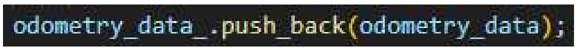


```
trajectory_builders_.push_back(absl::make_unique<CollatedTrajectoryBuilder>(
    trajectory_options, sensor_collator_.get(), trajectory_id,
    expected_sensor_ids,
    CreateGlobalTrajectoryBuilder2D(
        std::move(local_trajectory_builder), trajectory_id,
        static_cast<PoseGraph2D*>(pose_graph_.get()),
        local_slam_result_callback, pose_graph_odometry_motion_filter)));
```

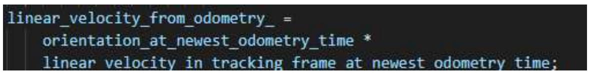


```
return trajectory_builders_.at(trajectory_id).get();
```

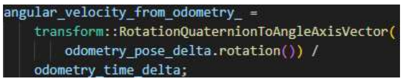




```
odometry_data_.push_back(odometry_data);
```



```
linear_velocity_from_odometry_ =
    orientation_at_newest_odometry_time *
    linear_velocity_in_tracking_frame_at_newest_odometry_time;
```



```
angular_velocity_from_odometry_ =
    transform::RotationQuaternionToAngleAxisVector(
        odometry_pose_delta.rotation()) /
    odometry_time_delta;
```