How to use?

The following steps are required in order to use the planning framework. For examples, see `/test`:

1. Start ROS (required for `ActionDispatcher`)
2. Start the mongoDB service. Under Linux: `sudo service mongod start`
3. Start a mongo client using pymongo
    a. `pymongo.MongoClient(host, port)`
    b. The standard host address and port is `localhost:27017`
4. Create an instance of the desired planner interface (eg. PANDAInterface). The constructor needs the following arguments:
    a. `kb_database_name`: Name of the mongo database name. Chose any name you like or use an existing database.
    b. `domain_file`: Name of the domain file (.hddl for the PANDA planner)
    c. `planner_cmd`: Shell command to execute the planner. "DOMAIN" and "PROBLEM" will be replaced by domain and problem files
    d. `plan_file_path`: directory where planner output will be generated temporarily.
5. For convenience, you can use yaml files to store the planner config. The yaml file are located in `/config`.
6. Use `planner_interface.kb_interface.insert_facts()` or `insert_fluents()` in order to add knowledge.
7. If planning is required, use `planner_interface.plan()` to start the planning. The function returns a bool value stating whether a plan was found as well as a list of actions. The parameters of the plan function are the following:
    a. `task_request`: this is a piece of heritage from the ropod project used to add more information to a task. At least the PANDAInterface does not use this parameter, so you can pass an empty TaskRequest() object.
    b. `robot`: Name of the robot (only used for logging)
    c. `task_goals`: list of task goals (htn tasks for the PANDA planner). List elements can be tuple, list or Task objects.
8. If a plan was found, the actions contained in the plan can be dispatched by `ActionDispatcher().dispatch_action()`, which takes an action and a timeout (in seconds) as arguments. The dispatcher will publish the `kcl_rosplan/action_dispatch` message and subscribe to `kcl_rosplan/action_feedback`.
9. If no robot is available to actually execute the action, the `action/action_execution_dummy.py` can be used in order to test the planning system. The dummy will just send a positive feedback to every received action (just execute it in a terminal).
10. **Important Note:** The planning framework will not update the knowledge with respect to the action effects. The responsibility for this is with the higher-level system.

How to adapt to a new domain?

- All predicates in the domain file have to be added to the (HDDL)PredicateLibrary, (HDDL)FluentLibrary or (HDDL)NumericFluentLibrary. Whether a predicate is represented as a Predicate or Fluent is purely a design decision. A Fluent for example is the preferred choice to represent a robot's position, allowing a robot only to be at one position at the same time.
- All actions in the domain have to be added to the ActionModelLibrary. If an action has to be modified, before being sent to the dispatcher, for example add some default parameters or

convert different actions in the domain to the same action with a different parameter (e.g. PickFromPlane, PickFromContainer), this can be done here.

Note: When editing the domain file, one has to be very careful for typos. In case of typos in the domain file the planner will (of course) return an error, but the error message is not very helpful in most cases.
It is not allowed to mix up tasks and methods. First all tasks have to be defined and then all methods.

Note on Predicates/Fluents: The terms 'Predicates', 'Fluents' and 'NumericFluents' can be confusing in the beginning. In general, they all describe the same thing namely a fact in the knowledgebase. Everything that is expressed by a fluent could be also expressed by predicates, but sometimes fluents are more convenient. Consider the following example: We want to express the current position of a robot in the knowledgebase. Using a Predicate, it would be something like RobotAt(?robot – RobotObject, ?loc – Location). Let's assume robot "Frank" currently is in the kitchen. We would then add to the knowledgebase: RobotAt("Frank","Kitchen"). Now the robot moves to the Livingroom. We add: RobotAt("Frank","Livingroom"). At this point, the knowledgebase is in an inconsistent state since the first Predicate we added is still in the knowledgebase. Physically the robot cannot be at two locations at the same time, so it is important that we delete all existing RobotAt-Predicates for "Frank" when we add a new one. To make things easier, the concept of Fluents was introduced. A Fluent represents a global state and it is not possible to have two Fluents with the same parameters. For the robot's position, it would be for example RobotAt("Frank") = "LivingRoom". When updating this fluent, the old value is overwritten and there is no need to delete old values. The concept of Fluents is only used inside the knowledgebase. When communicating with the planner (i.e. creating a problem file), all Fluents are converted to predicates since Fluents do not exist in PDDL or HDDL. The difference of Fluents and NumericFluents is just the fact that the latter represent numeric values (e.g. weight of an object) and thus mathematical operations can be applied to them. In the end it is a design decision whether to represent a Predicate in the domain as a Predicate or as a Fluent in the knowledgebase.