Building A Domain

Domains are responsible for creating skills (i.e. declaring tasks). They extract parameters from the matched query and deploy a corresponding skill that must fulfill the user's query. The process is complex and every step of building a new domain is documented here.

Execution

Step 1 - User Input: The user produces a string, which from now on we will call "query". When this happens the U.I. (i.e. user interface) calls "processQuery".

What exactly happens inside of "processQuery"?

1. Every registered domain shows the assistant a list of patterns that they require in the query. These patterns we will call speech patterns.
2. Then the assistant tries to match the query with every speech pattern. Some of them will match and some of them will not.
3. The assistant now discards all the domains whose speech patterns did not match.
4. From the domains that are not discarded, the assistant will rank each one of them based on a ratio: A/B , where A is the number of tokens in the query that matched with the speech pattern and B is the number of tokens in the query.
5. Then, the assistant picks the domain that reported the highest ratio. Naturally, it also picks the speech pattern that corresponds to this ratio.
6. Finally, the assistant calls "dispatchSkill" on this domain.

Step 2 - Declare Task: The domain is now responsible for using the data so far (i.e. the matched speech pattern and the query) to generate a background task that fulfills the request of the user. We will refer to this background task as skill.

Note: From this point forward there is an agreement between the domain and the assistant that you must be aware of. After the assistant has picked a domain, the domain is obligated to fulfill the user's request. Why? Because the domain is supposed to show the assistant only those speech patterns that ensure the user has provided all necessary data to process their query. Otherwise, the domain is unfit for the user's request and thus should not have been picked.

The domain is provided with a MatchedSequence object, which contains a reference to the pattern that was matched, a reference to the query, and a sequence of pairs. For each pair, the first element is a slot in the pattern, and the second element is a list of matched tokens from the query.

Once the domain has extracted all the data necessary, it can decide what skill to generate (i.e. domains can dispatch different skills - only one per request)

Step 3 - Start Task: Now the assistant receives the skill generated by the domain and runs it in a separate thread (background thread) and keeps a registry of running skills. Once a skill finishes the corresponding task, it is removed from this registry.

Building A Domain

Step 1 - Declaring patterns: The first step is to declare all speech patterns that belong to this domain. For example, in a domain for taking photos (i.e. Photo Domain) we should define all speech patterns that we think the user may try when requesting to take a photo. How many ways are there of asking this? Way too many, so we only define a few speech patterns that are somewhat resistant to minor variations and assume a narrow set of instructions.

For instance, what ways of taking a picture do we want to support? Let's say we want to support taking a picture immediately and taking a picture after x seconds. Then we only have to think of speech patterns that correspond to this narrow set of supported skills.

In my opinion (which is subjective) these speech patterns should be:

- <photo, selfie, picture>
- <photo, selfie, picture> <#:3> <in, after> <param:int>
- <param:int> <#:4> <photo, selfie, picture>

Why these speech patterns?

- The first speech pattern will match with anything that contains either one of those three keywords (i.e. selfie, photo, picture).
- The second patterns will match with a wide range of variation where the user asks for a picture, but wants the assistant to wait a few seconds.
- The third speech pattern is similar to the second one, but assumes the user will specify the waiting time before asking for a picture.

Note how I limited the length of the blank slots in the last two speech patterns. This is like that for a very important reason. Speech patterns should only try to match with the minimum information necessary. This way, if two domains happen to match with the same query, only the one that truly uses the majority of the information in the query will be selected.

Examples:

- "Take a picture of me after 5 seconds" - Match
- "Please, would you make a photo of me, but wait 4 seconds?" - Not Match
- "Take a selfie of me in 10 seconds" - Match
- "Take a selfie of me, and wait 10 seconds" - Not Match

Some of these queries should match, but they don't. This is a compromise, our A.I. won't be able to understand everything. So, we should focus only on the most common utterances. However, for this particular case, the fix is easy. Simply increase the maximum length of the blank slot.

Note that if we make the blank slot infinitely long (i.e. <...>), it will match with any query that contains "photo"/"selfie"/"picture", "in"/"after", and followed by any integer. This is really bad.

This is the step that requires the most planning.

Step 2 - Extracting Data: There is a lot of freedom in this part. Here you are expected to use MatchedSequence to determine what the user wants. However, there are two constraints:

1) The domain should not block the main thread. Anything that can block, should be part of the skill instead.
2) Do not reference global variables from the skill, skills are supposed to be thread safe. So, if the user spams the assistant with the same query over and over and the same skill is dispatched over and over, the app should not crash.
3) If skills wait for some time, do it in such a way that if the assistant calls Thread.interrupt on said skill, it stops processing. Why? Otherwise the assistant would have to wait until the skill stops before closing the application.

Once the domain has determined what the user wants, it can return a new (and custom) Skill object to the assistant.

Note:

- Skills are akin to subroutines and domains are akin to programs. So, if you want to deploy domain specific GUI, like a prompt or something, you can do it. Just make sure the domain or skill does not interfere with the main thread. (i.e. you can start your own swing app or javafx app if you want, just don't mess with the main thread)

- Skill should never invoke System.exit. Why? Because before closing the assistant, it has to notify all running skills to stop. If this is done incorrectly, huge memory leaks will happen. Exiting the assistant app should be done through the GUI only.
    - If you want your skill to be able to close the assistant, create a new messageType and coordinate with the GUI team how to handle this new type of message. (of course make sure this message is not already a feature)