I’m looking through the code for the Little-JIL scenario, and I know there are things that I’d like to play around with. As I look through the code itself, I’m going to jot down some notes here about where I’ve found certain methods!

**Scenario.java** → This class sets up a lot of the classes needed to run the scenario! Of importance are a couple of the following pieces:

* Default\_Scene\_ID\_List → this stores all of the different names of the scenes. Some of the default ones include “project task assignment”, “choose initial action”, etc. The names of these must match the same names in the Little-JIL tree leaves, so if I add / change anything here, I need to change them in the diagram, too.

**Scene.java** → This is the class that acts as a base-level scene! Its declaration has a generic type, C, which is some kind of SceneInfo type that’s used for the scene’s content. There are a couple of different subclasses of Scene, all which use a different type of SceneInfo for the generic content type of Scene.

* **ImageScene.java** → This one will presumably set up an image scene!
* **VideoScene.java** → Naturally, this one will set up a video!
* **QAndAScene.java** → This one is a scene where the player can enter in a response to a couple of questions. I took a decent amount of time to try and figure out how these work - it seems like the answers are pulled in the decode( ) method, and then stored in each QAndAItem object that was passed to the scene in the first place.
* **MakeDecisionScene.java** → This is a scene where a player makes some sort of decision from a drop-down list. It extends Scene with some MakeDecisionInfo object, similar to the QAndAItem object; the MakeDecisionInfo contains the list of decisions, and stores players’ decisions in its selectedOption\_ field, which you can access through the getSelectedOption( ) method.

**ConsequencesDocumentTemplate.java** → This one seems to handle choosing the consequences from a list of consequences. A list, consequenceSections\_, is built up throughout the runtime of the program; then, when someone passes an Information and Action object to the getConsequencesSection( ) method, a specific consequence will be returned if its Information and Action objects satisfy that consequenceSection’s ConsequenceFilter’s accept( ) method. The concept of ConsequenceFilters is pretty cool, so I’m going to list a couple that I’ve found so I can understand how they work:

* **ModerateConsequence.java, TerseConsequence.java, and VerboseConsequence.java** → All of these are different ConsequenceFilters that Purity wrote in order to make her different scenarios work! Depending on what Information and Action are passed to the accept( ) function, different filters will activate. The Information being looked at is the number of conversations had, and the action taken is whether or not the player chose to delay the project.
* **AllConsequence.java, AndConsequence.java, OrConsequence.java** → These are all consequence filters that combine multiple consequence filters! As soon as I saw these, a lightbulb went off in my head about all of the different things you could string together for consequences.

**Information.java** → This class stores a couple of pieces of information about the state of the game, and has the methods to update them. Here are some of the components that it has:

* stakeholderToHadConversation → This is a LinkedHashMap of (String, Boolean) pairs; each of the strings represents a stakeholderKind, which is the name of the
* type of conversation that’s had.
* isDone → This is a Boolean which determines if all of the conversations have been gone through.
* getConversationCount() → This one will return the number of conversations that have been had (it just returns the size of stakeholderToHadConversation)

**Company.java, Project.java →** These classes will hold some information about the company you work for, and the project that you’re working on. I can see the strings where they’re used, but I forget when those particular strings come in in the actual scenario. It’s a smart idea, though - it makes the scripts much more modular!