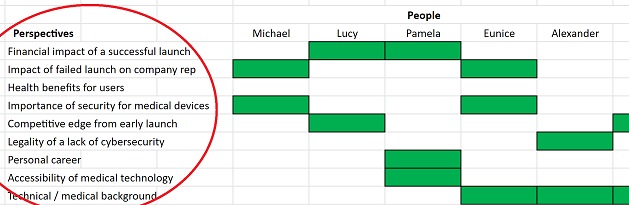
### **Classes I’ve Created**

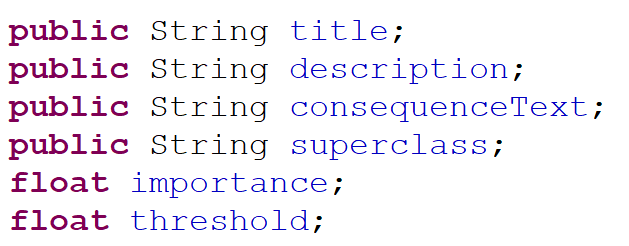
*To implement my scoring system, I needed to create a couple of new classes. I’m going to give a brief overview of the purpose of each class. Then, in the “Configuring Your Scenario” section, I’ll talk about how you’ll use these classes in order to suit your scenario to the system!*

**Perspective.java**

This class represents one of your scenario’s perspectives! If we made a perspective coverage matrix, these would be the row headers of the matrix.



The Perspective class is pretty simple: it just serves as a container for a number of instance variables, each of which holds information about that perspective. The instance variables are as follows:

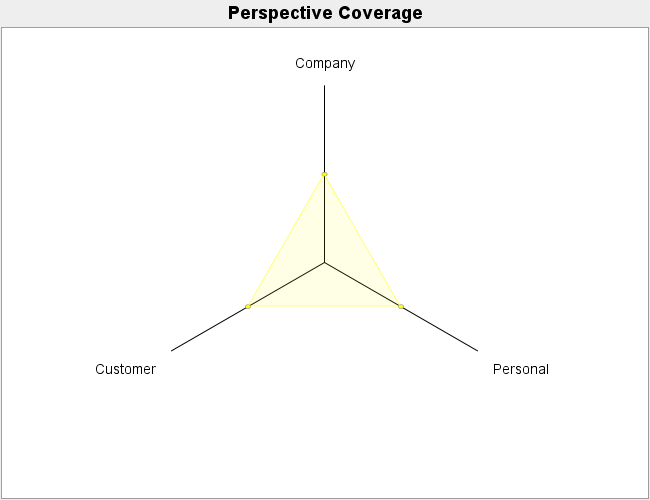


The ***title*** String is a small description of the perspective! (So, for the “Accessibility of medical technology” perspective in the matrix above, you’d probably call it something like “Accessibility”.) These titles will be used to identify Perspectives, and as labels for the eventual spiderweb charts at the end.

The ***description*** String is a longer description of what the perspective is - this might be the full text in the matrix above! (This is used later on, when building the consequences document, as headers for sections about each perspective.)

The ***consequenceText*** String is text that will be displayed in the consequences document if that perspective is not covered enough by the player. (The threshold that determines what constitutes as “enough” is the float, ***threshold.*** Make this between 0 and 1!)

The String ***superclass*** is way of grouping different Perspectives into larger categories; so, later on in the scenario, during the generation of the consequences document, most of the different perspectives will be grouped into different “superclasses”. Then, these superclasses will be used to make one of the spiderweb charts; an example is below:



*In this spiderweb chart, the three superclasses are “Personal”, “Customer”, and “Company” - most of the different perspectives in the matrix were factored into one of these superclasses.*

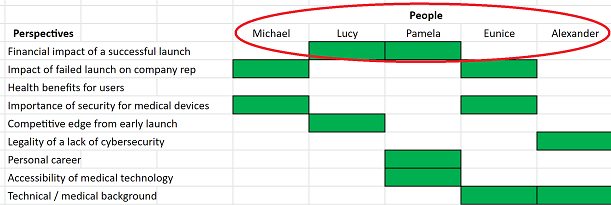
Finally, the ***importance*** Float determines a “level of importance” for the perspective. These importance levels ought to be between 0 and 1, and they’re used to determine the “value” of having a conversation with a Stakeholder; basically, this equation is as follows:

That equation might be a little confusing right now, so an example might serve well. Let’s say we have a stakeholder, Jim, who has two covered perspectives: PerspectiveA and PerspectiveB. He has 1 coverage point for each of them; PerspectiveA has an importance of 1, whereas PerspectiveB has an importance of 0. Then, Jim’s total value will be 0.5! If, say, another stakeholder, Nancy, covered PerspectiveA and PerspectiveC with 1 coverage point for each, and PerspectiveC had an importance of 0, then Nancy would have a value of 1. (In this way, conversing with Nancy ought to have more value for the player, as she covers perspectives that have been deemed as more important than Jim’s.

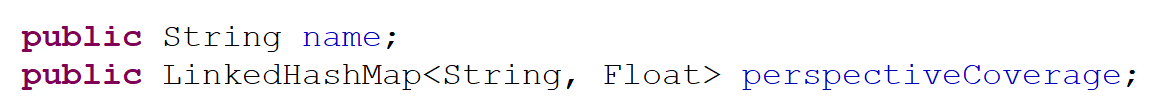
Generally, these importance values ought to be set in such a way that reflects the scenario designer’s intentions. If a perspective reflects something fairly important to understanding the ethical dilemma, it ought to have a higher importance score (and vice versa).

**Stakeholder.java**

This class represents one of your scenario’s stakeholders! If you’ve made a perspective coverage matrix, then these stakeholders would be the column headers:



The Stakeholder class is *much* more simple than the Perspective class - it contains the following instance variables:

****

The ***name*** String indicates the name of the Stakeholder! This field is used to distinguish different Stakeholder objects from one another.

The LinkedHashMap<String, Float>, ***perspectiveCoverage***, is used to store that particular stakeholder’s perspective coverage points! A perspective coverage point indicates that a stakeholder has covered that particular topic in their conversation. In the above matrix, for example, you could assign Michael a coverage point for both the “Fail Reputation” and “Security” perspectives by calling michael.setPerspectiveCoverage(“Fail Reputation”, 1f) and michael.setPerspectiveCoverage(“Security”, 1f). (You could also check Michael’s perspective coverage map for a specific perspective by calling michael.getPerspectiveCoverage(\_\_\_\_\_), where the \_\_\_\_\_\_ is a particular perspective name.)

**SpiderChartBuilder.java**

This class is used to generate the spiderweb chart that’ll eventually be used in the consequences document! In order to actually make this class, I needed to download a couple of JAR files: [JFreeChart](http://www.jfree.org/jfreechart/) (which handles the chart creation) and [JCommon](http://www.jfree.org/jcommon/%5C) (a dependency of JFreeChart). There are three main methods within this class: ***createSuperclassDataset*** (which prepares the data for the superclass chart), ***createSpecificSuperclassDataset*** (which prepares the data for a spiderweb chart on a *specific* superclass), and ***createChart*** (which actually creates and saves the chart as a PNG!) Currently, the charts are being saved in MedicalDevicesCyber/End/Images - you can change this path near the end of createChart( ).

**ConsequencesBuilder.java**

This is the class that’s being used to generate the PDF for the consequences document! In order to actually make this class, I needed to download the JAR file for [PDFBox](https://pdfbox.apache.org/), a library dedicated to the generation of PDFs.

This class is by far the messiest, least generalized of all of the ones I added. The library doesn’t have the most elegant creation tools - you need to specify the positioning of certain lines of text and images, and you need to break up a multi-line String into separate Strings in order to actually have it display without overflowing on the page. I’d written a couple of different helper methods in order to make document creation a little easier (like ***lineBreaker***, which separates a String into a Set<String> with one string per line), but it’s still a little more custom built than I might have originally liked.

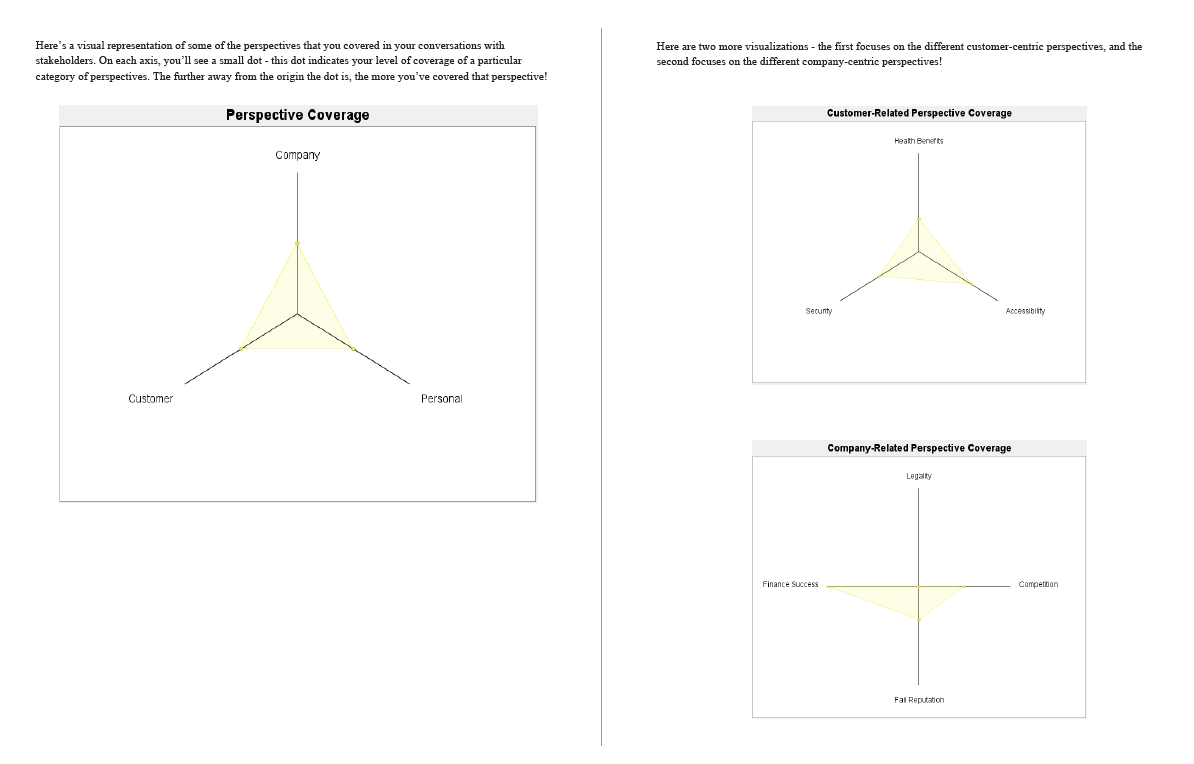
Additionally, since the ConsequencesBuilder class contains a lot of logic about a player’s decisions throughout the scenario, I’ve basically bypassed the use of ConsequenceFilters in my scenario. (I couldn’t figure out an elegant way to actually do this with the filters, so I just put all of the logic in one place - this class - and called it a day.)

The resulting consequences document is made of three different parts! The first page deals with the player’s perspective coverage - it contains a brief description of the scenario and its quandary, as well as a list of perspectives whose thresholds weren’t cleared by the player.



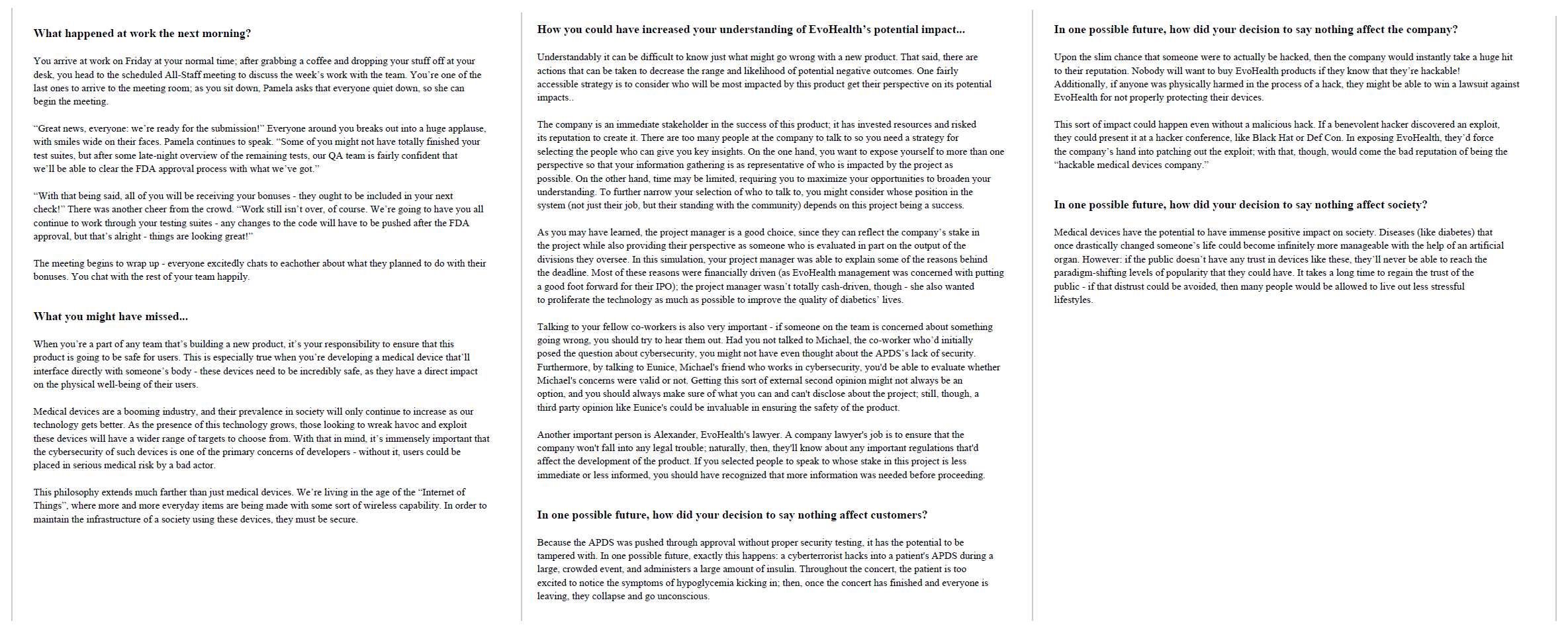
*An example of the perspective coverage page.*

The next two pages deal with the visualizations! Here, the class uses the SpiderChartBuilder class to make three spiderweb charts. One of them is a visualization of my scenario’s three superclasses: Company, Customer, and Personal. Then, the other two are more spiderweb charts that drill into the perspectives covered within both the Company and Customer superclasses.



*An example of the two visualization pages.*

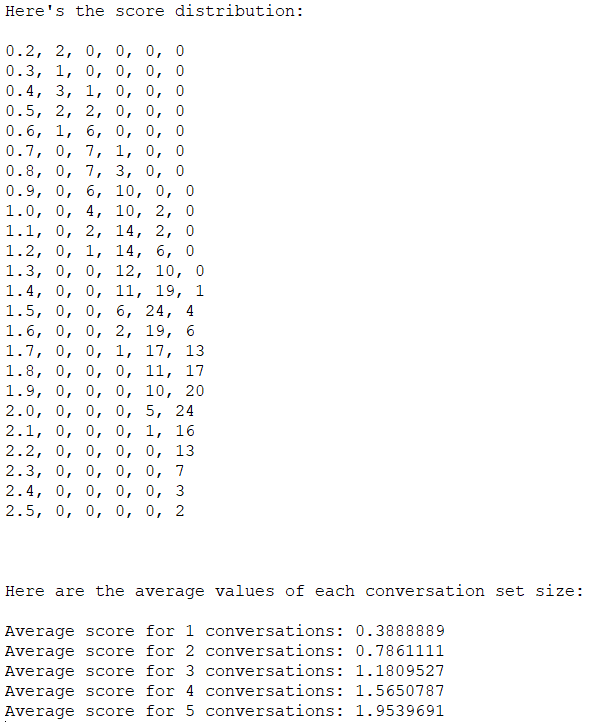
Finally, the last section contains three pages of text, explaining the rest of the scenario’s narrative and demonstrating the impact of the player’s decision.



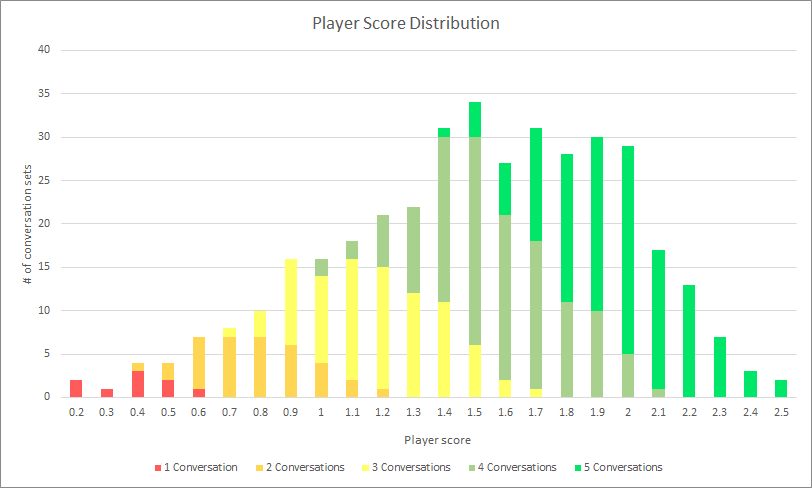
*Here’s an example of the three end pages!*

**ScoreTest.java**

This method will print out a couple of statistics about a particular configuration of a scenario’s Perspectives and Stakeholders. If you run the stats( ) method, you’ll receive an output like so:



The first bit, which is a list of values in comma-delimited format, can be used to produce a graph like this:



In order to do that, just save that text as a .csv, open it in Excel, and then align it in a chart like the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Score** | **1 Conversation** | **2 Conversations** | **3 Conversations** | **4 Conversations** | **5 Conversations** |
| 0.2 | 2 | 0 | 0 | 0 | 0 |
| 0.3 | 1 | 0 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... |

Then, highlight this data and insert a Stacked Column Chart, and viola!

### **Configuring Your Scenario**

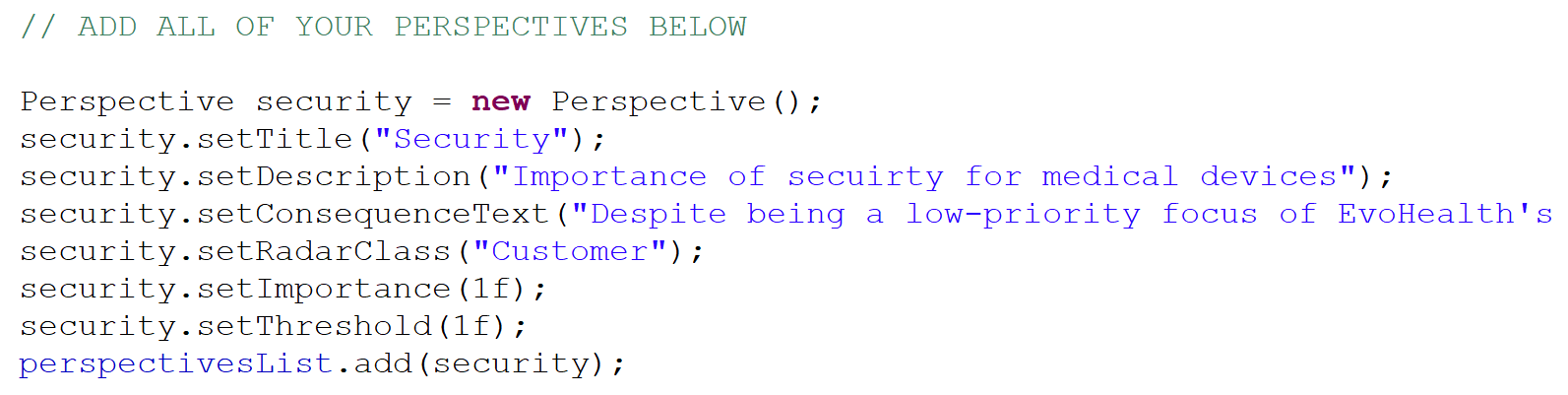
*In order to set up your scenario to be used with my scoring system, there are a couple things you’ll need to do! This section will explore all of those things.*

**Preliminary Setup**

Before jumping into the code, you’re going to want to ensure that you have a perspective coverage matrix defined! This will give you a plan for suiting your scenario to the scoring system. In order to ensure a good balance, make sure each perspective is covered at least once by a stakeholder, and that each stakeholder examines a unique set of perspectives!

**Setting Up Perspectives**

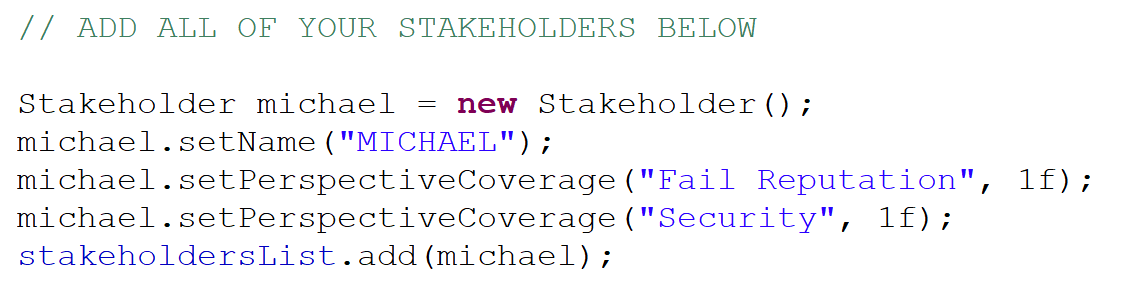
The first thing you’ll have to do to configure your scenario is declare the Perspectives that are being used! This will be done in the constructor for your scenario class, right after the call to super( ). Below, you’ll see one of my Perspective declarations:



The process is fairly routine - you just need to create a new Perspective object and then set all of the various instance variables. Then, when you’re done, you can call perspectivesList.add( ) to push it to the scenario’s list of perspectives!

**Setting Up Stakeholders**

This step is fairly similar to the Perspectives setup; immediately after declaring all of them, you’ll need to declare your Stakeholders! With each Stakeholder, you need to set a name, and then specify the perspective coverage points for that particular stakeholder!

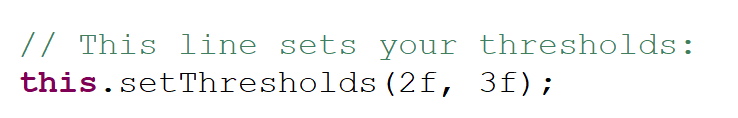


Here’s one important caveat about the Stakeholders: the order you declare them in needs to match the order of the Stakeholder names in StakeholderKindEnum, which *also* needs to match the order of the documents that are added into the conversationSceneList! I wasn’t sure how to avoid using the StakeholderKindEnum, as it’s still used for one of the calls to MakeDecisionScene (which happens *before* the declaration of the Stakeholders).

**Setting Your Score Thresholds**

Once you’ve finished adding your Perspectives and Stakeholders, you need to define the different player score thresholds! The player’s score the sum of the values of all of the Stakeholders that they talked to throughout the simulation. In my schema, there are three different “grades” for players: Poor, Good, and Great. These grades are never directly indicated to the player, but certain pieces of text within the consequences document (as well as the color of the spiderweb charts) will change depending on the grade.

If their total score is less than the poor threshold, they’ll receive a Poor grade. (This is indicated by red spiderweb charts!) If it’s greater than the poor threshold, but smaller than the good threshold, they’ll receive a Good grade. (This grade will make the spiderweb charts yellow!) Finally, if the player’s score is larger than the good threshold, they’ll receive a Great. (This will change the color of the charts to green.)



*You can set the thresholds using this.setThresholds( ) - the first number is for poorThreshold, while the second one is for goodThreshold.*

**Editing ConsequencesBuilder**

The final step in the configuration of your scenario is editing the ConsequencesBuilder class. Unfortunately, since this class is fairly custom-built to the specifications of my text, it’s hard to write a general guide for configuring this. So, instead, I’ve tried to leave comments in the class itself, explaining what each portion of the createPDF( ) method is doing! If you want to create a consequences document with the format described earlier, I’d look through the comments I left on this class and try to swap out some of my text for your own.

### 

### **Changes to Pre-Existing Classes**

*In order to better implement my scoring system, I needed to change some of the pre-existing classes. Here’s a list of all of the larger changes I’ve made, along with some justification for them - hopefully, these ought to help you make further changes if necessary! (Apologies if I’ve forgotten to list anything - I made this in retrospect after the changes were made, so certain things might have slipped through the cracks.)*

**Information.java**

* I added some instance variables:
  + Action actionTaken - this is meant to store the player’s final decision (i.e., whether or not to delay the project). This way, the ConsequenceBuilder could use this action to decide which blocks of text to use!
  + Set<Stakeholder> stakeholdersList and a Set<Perspective> perspectivesList, both which hold all of the stakeholders and perspectives for the scenario!
  + Float poorThreshold and Float goodThreshold, which are variables that determine the “grade” of the player’s perspective coverage
* I altered the Initialize( ) method a little bit - it now takes a Set<Stakeholder>, a Set<Perspective>, and a Float[] (for the thresholds) as arguments, and it’ll set the respective instance variables to each of these arguments!
* I added a couple of methods related to calculating scores for the player, including:
  + getPlayerScore( ), which determines the player’s total score
  + getStakeholderValue( ), which determines the value of a single given stakeholder
  + getCoverage( ), which determines how much of a perspective a player has covered (this is given as a 0-1 float, 0 being 0%, 1 being 100%)
  + getSuperclassCoverage( ), which determines how much of a given superclass of perspectives a player has covered (using the same scale as getCoverage( ))

**Scenario.java**

* I added an Information instance variable called “playerInfo” - this is meant to store the player’s active Information object so that it can be passed off to the ConsequenceBuilder during the setting of the consequences scene.
* I’ve added a pair of instance variables: a Set<Stakeholder> called stakeholdersList, and a Set<Perspective> called perspectivesList. (Each of these has a respective getter and setter method!)
* I added two Float instance variables: “poorThreshold” and “goodThreshold”, each of which share a getter and a setter. Basically, these are used to determine whether or not the player has had “poor” perspective coverage, “good” perspective coverage, or “great” perspective coverage!

**GameSystemItemHandler.java**

* I updated the initialization of the initialInformation parameter by adding arguments as per my changes to the Information’s initialize( ) method
* I added some statements in the “determine consequences” step, namely
  + Updating the playerAction within the information
  + Updating the scenario’s information parameter

**MakeDecisionScene.java**

* I added another version of the constructor! Instead of using an Enum to generate the different decisions, this one uses a Set<String>. This way, I could use the names of the Stakeholders I added in order to populate the DecisionScene. Unfortunately, I couldn’t figure out an elegant way to actually use this, as in the TrevorsScenario.java class, the DecisionScene is set up before the Stakeholders are declared.