SoftGel Pills 2:

Pill Production and Variations

Table of Contents

From the Execs

Thank you for making this excellent foundation. However, we do not sell generic AcheAway or Dreamly, we sell adult and child variations. Please create those as well as a store for us to sell the pills. We will provide you with a sample client demo to show you how we want it to work. Here are the specs for our child and adult pills

	Strength (mg)	Size (mm)	Color
Adult Dreamly	5.20	12.24	tan
Adult AcheAway	825.00	8.50	white
Child Dreamly	1.25	4.5	fuchsia
Child AcheAway	415.00	3.25	cyan

From your Manager

The executives are asking for a lot, probably more than they realize. If we are producing all of these pills, we should really structure our code better. The pills should not be in charge of creating themselves, so let's make a factory. Since we have two families of pills, child and adult, we should use the abstract factory pattern. Our factories should use the template method so both factories use the same process and we do not need to duplicate code. Let's also use the strategy pattern for the casings, solutions, and actives. They will probably ask us to add more types of pills, so we want these to be convenient and reusable. I will provide you with a UML for how this should be structured. It is a lot, but most of the classes will be very simple since we are leveraging abstraction and polymorphism as much as we can.

Project Setup

Repository

You will be using the same repository that you used in Part 1. Make sure you have made an Introduction branch off of main before you start. You should update the demo with the demo provided by the executives.

Group

This project will be split up into three sprints. The product backlog is provided for you. Each sprint will be 1 week long. For each sprint you should do the following things.

1. Create a file in docs named SprintX.txt where X is the sprint number (2, 3, and 4)
2. At the beginning of the file, write Sprint Planning:
3. Make a sections named Sprint Backlog
4. Decide on what items from the product backlog you want to work on this sprint and write them down under Sprint Backlog.
5. Create your subteams like before (these can change each sprint).
6. Assign backlog items to the subteams. You may decide some items should be assigned to one person or some items should be assigned to everyone, but most of them should be assigned to a subteam.

7. At the end of the sprint, do a sprint review. Make a section in the document called Sprint Review (you could also make a separate file if you want these files to be smaller).
8. In the sprint review, discuss your product at this point. Does it work? Are you happy with the code you have created?
9. Then create a Sprint Retrospective section and perform a reflection in the same way you did in the Introduction project. However, this time you will be immediately implementing any changes you want to make for the next sprint.

When you are selecting items from the product backlog to put in your sprint backlog, you can split them into smaller tasks or combine them into larger tasks as you see fit. It is a good idea to assign values to them (e.g. planning poker) so that you can evenly distribute the work. The way that your team chooses to do this is up to you, but you should document it in Sprint Planning.

Product Backlog

- Update GelCap
- Update the tests for GelCap
- Update Dreamly and AcheAway
- Update the tests for Dreamly and AcheAway
- Create child and adult versions of Dreamly
- Create tests for the child and adult version of Dreamly
- Create child and adult versions of AcheAway
- Create tests for the child and adult versions of AcheAway
- Create the CasingGenerator interface and the two types of Casing Generators
- Create tests for the two types of casing generators
- Create the SolutionGenerator interface and the two types of Solution Generators
- Create tests for the two types of solution generators
- Create the ActiveGenerator interface and the two types of Active Generators
- Create tests for the two types of active generators
- Create the GelCapRecipes class
- Create tests for GelCapRecipes
- Create the GelCapFactory class
- Create tests for GelCapFactory
- Create the Adult and Child factory classes
- Create tests for the Adult and Child factories
- Create the Store
- Create tests for the store

Detailed Class Descriptions

For all classes, see the UML diagram for basic structure and information

GelCap

- Update to match UML
    - Update constructor
    - Remove unused methods

AcheAway

- Update to match UML
    - Update constructor
    - Remove unused methods
    - Make Abstract

Dreamly

- Update to match UML
    - Update constructor
    - Remove unused methods
    - Make Abstract

ChildAcheAway, ChildDreamly, AdultAcheAway, AdultDreamly

- Subclasses of AcheAway or Dreamly respectively.
- Should have constants for strength, size and color

- Should have a constructor that just calls the super constructor

GelCapRecipes

- This class is used entirely statically
- There are three HashMap constants that the factories will access.
- In order to set these constants statically, each has a corresponding private static method that creates and sets the constant. These methods should be called when the constants are declared.
- Each hashmap should have two keys: "dreamly" and "acheAway"
- The values should be the corresponding type of generator. See the instructions from part 1 for which generator should be applied which pill.

CasingGenerator, SolutionGenerator, ActiveGenerator

- Interfaces that each have just one method as defined in the UML

The 6 Generator Classes

- Each should implement its respective interface
- Just needs to override the method from the interface
- This method should have several print statements and then return the corresponding String. Here is what should be

printed and returned for each class. For the active classes, the amount passed to the method is what should be printed for the %.2f variables

GelatinCasing	Mixing gelatin, water, opacifier, and glycerin ... \n Shaping ... \n Returning gelatin casing ... \n	gelatin
PlasticizerCasing	Mixing starch, water, opacifier, and glycerin ... \n Shaping ... \n Returning plasticizer casing ... \n	plasticizer
SalineSolution	Mixing salt and water ... \n Verifying ratio ... \n Returning saline solution\n	saline
OilSolution	Extracting pill-sized quantity of oil ... \n Returning oil solution\n	oil
AcetaminophenActive	Acetylating para-aminophenol with acetic anhydride\n Carefully extracting %.2fmg of acetaminophen\n Returning %.2fmg of acetaminophen\n	acetaminophen
ZolpidemActive	Opening secure storage area ... \n Carefully extracting %.2fmg of zolpidem\n Returning %.2fmg of zolpidem\n	zolpidem

GelCapFactory

- Abstract class from which our concrete factories will inherit.
- produceDreamly, produceAcheAway

    - These methods should be final. They are our template methods
    - They should start by printing "Creating a X pill ... \n" where X is either Dreamly or AcheAway
    - Then they should create a pill using constructDreamly or constructAcheAway and pass it a casing, solution, and active that it retrieves from GelCapRecipes.
    - The generateActive method will require a strength. Use the abstract getXStrength method to pass a value to it, where X is either Dreamly or AcheAway.
    - Then it should call qualityCheck
    - If qualityCheck passes, it should print "Returning a good X GelCap Pill\n" where X is either Dreamly or AcheAway. It should then return the created pill.
    - If qualityCheck fails, it should print "Error during X production. Returning null.\n" where X is either Dreamly or AcheAway. It should then return null.
- qualityCheck
    - private and final
    - This should first print "Performing quality check ... \n"
    - Then it should use a random number generator (such as Math.random()) to decide whether or not the pill passes. It should pass 90% of the time.
    - If it passes, print "quality check passed\n" and return true.
    - If it fails, print "quality check failed\n" and return false

- constructDreamly, constructAcheAway, getDreamlyStrength, getAcheAwayStrength
    - These are all abstract
    - Nothing should be done aside from defining them

AdultGelCapFactory, ChildGelCapFactory

- Inherit from GelCapFactory
- They only need to override the four abstract methods.
- constructDreamly, constructAcheAway
    - Print "Constructing X version of Y\n" where X is adult or child and Y is Dreamly or AcheAway
    - Then it should return a new object of the corresponding class (AdultAcheAway, AdultDreamly, ChildAcheAway, ChildDreamly).
- getDreamlyStrength, getAcheAwayStrength
    - These methods can simply return the constant from their corresponding pill type. For example, getDreamlyStrength in the AdultGelCapFactory would return AdultDreamly.STRENGTH.

SoftGelPillStore

- This will likely be the most challenging class to write.
- constructors
    - There are 4 constructors. The one that takes input and output should simply set the corresponding fields to those parameters

    - The other constructors should use the this() keyword to call the 2-arg constructor.
    - If they do not have an input argument they should pass new Scanner(System.in)
    - If they do not have an output argument they should pass System.out
- setOutput, setInput, getOutput, getInput
    - These are basic getters and setters
    - They simply need to return or set their corresponding field
- logIn() noArg
    - First this method should ask "What is your name?"
    - It should then get input using the input field and assign it to a String variable.
    - Then it should ask "What is your age?"
    - It should then get input using the input field and assign it to an int variable.
    - It should re-ask until the user enters a valid age and should not crash on bad inputs. Negative ages and non-numbers are both invalid
    - Finally, it should call the two-arg logIn method with the name and age it received from the user
- logIn(name, age) 2-arg
    - Sets the customerName and customerAge fields
    - Sets the factory to either a child or adult factory based on the customer's age. If they are under 18 it

should be a child factory, otherwise it should be an adult factory
    - It should then set isLoggedIn to true and create a new ArrayList for currentOrder
- order
    - If the user is not logged in, print "You must log in before you can order.\n" and return.
    - If the user is logged in, print "Hello, %s. What would you like to order?\n" where %s is the customer's name
    - Print a menu in the following form

        Options:
        1) Dreamly
        2) AcheAway
        3) Cancel
    - Get input from the user and if it is not a 1, 2, or 3, print "Please enter a 1, 2, or 3\n" and print the menu again. Continue looping until the enter a valid selection. This should not crash on bad inputs
    - Once you have the input, use the factory field to produce the type of pill selected and store it in the currentOrder ArrayList.
- checkOut
    - If the user is not logged in or has no pills in their current order, print "You need to log in and order before you can checkout\n" and return null
    - Otherwise, print "Thanks for shopping!\nHere is your order\n"

    - Then print out every pill in the order (using toString, not description).
    - Use the toArray method of ArrayLists to create an array with the pills.
    - Call the clear method on the currentOrder
    - Return the array of pills
- logOut
    - If the user is not logged in, print "You are not logged in." and return false.
    - If the current order has at least one pill in it, print "You have an order that you have not checked out. Are you sure you want to log out? (y/N)"
    - If the user enters a y, then continue. On any other input, return false. Again, do not let the program crash on bad inputs
    - Finally, set isLoggedIn to false, the current order to null, the customer's name to the empty string, the customer's age to -1, and return true.

Detailed Test Descriptions

Each of the three classes should have a corresponding test file in src/tests/pills.

GelCapTest

- Modify the existing tests to work with the changes made to GelCap
- Remove the tests that are no longer necessary
- You will no longer need to redirect System.out

AcheAwayTest & Tests for Child & Adult Versions

- Since AcheAway is abstract now, you will need to create a private inner class to mock it just like you did with GelCap
- You will no longer need to redirect System.out
- Remove tests that are no longer relevant
- Update tests to work with the new constructor
- Add new constants for ADULT_STRENGTH, ADULT_SIZE, ADULT_COLOR, and repeat for the child versions.
- Add new test fields for adult and child dreamly
- initialize these test fields in BeforeEach
- For each test, add two assertions, one for the adult version and one for the child version. Example for testSize
  assertEquals(TEST_SIZE, dreamly.getSize());
  assertEquals(ADULT_SIZE, adultDreamly.getSize());
  assertEquals(CHILD_SIZE, childDreamly.getSize());

DreamlyTest

- This test file should be updated in the same way as the AcheAway test file.

SolutionTest

- This test file will test both solution classes.
- We do not need to test the interface itself
- You will need to redirect System.out
- There should be two tests: testOil and testSaline
- Each test should have two assertions: assert that the correct value was returned and assert that the correct text was printed

ActiveTest & CasingTest

- These test files are structured the same way as SolutionTest.
- See those instructions for details.

GelCapRecipesTest

- You should have three methods here: testCasings, testSolutions, and testActives.
- Each method has two assert statements, one for Dreamly and one for AcheAway.
- Each assert statement should check that the generator returned from the HashMap is of the correct type.

- ie: GelCapRecipes.ACTIVES.get("dreamly") should be a ZolpidemActive object

GelCapFactoryTest

- First, you will need a mock class just like in GelCapTest
    - For constructDreamly and constructAcheAway, print "X called" where X is the name of the method. Then return null OR a DreamlyMock / AcheAwayMock
    - For getDreamlyStrength and getAcheAwayStrength, do the same type of print statement as the previous methods and return 0.
    - We will be using these print statements as a hacky way to spy on these methods.
- produceDreamly and produceAcheAway
    - Test that the print statements are correct (this involves the print statements that are in the mock)
    - This test is difficult because there are two possible scenarios depending on if qualityCheck passes or fails.
    - Use a loop to call the method at least 100 times. Keep a counter of whether it succeeded or failed and save one of each response type. assert that both return values are of the correct type and assert that approximately 10% of the responses are failures (you will want to specify an acceptable range). The fail() function should be very useful here. The correct types for the return types will depend on your implementation in the mock class.

    - Note that the output will include the output from the generators. The best thing to do would be to strip this output from the output and just test the print statements from the factory. This output comes from the construct method. You can take it out of the output by removing everything between "Creating a X pill ... \n" and "Performing quality check ... ". Regular expressions and substring would be good methods of doing this. The string will be in the form <FirstPrint><PrintsFromGenerators><EndingPrints>
    - You simply want to remove <PrintsFromGenerators> from the string using whatever method you prefer. If you prefer to include the prints from the generators, you can put them in your test.
- qualityCheck
    - This method is private so you cannot test it. The test in produceDreamly and produceAcheAway effectively tests it already.
- No testing the abstract methods

AdultGelCapFactoryTest, ChildGelCapFactoryTest

- Test that the getters for the strengths return the correct numbers.
- Check that the construct methods ...
    - Print the correct output
    - Return the correct type of pill. You can use the instanceof keyword to check this

SoftGelPillStoreTest

- Get ready, there's a lot to test here.
- Dealing With I/O
    - There are multiple ways to do this and I will leave it up to you.
    - The Store accepts a PrintStream and a Scanner so you can provide those instead of redirecting System.out and System.in
    - My method was to use the one-arg constructor to give it a PrintStream with a BAOS and then used setInput in my methods that needed to test input and gave it Scanner which I passed a String of the input.
- Constructors
    - I would recommend testing all of the constructors in one test to reduce the size of the test file.
    - For the two-arg constructor, create a PrintStream and Scanner variable, call the constructor, and use the getters to check that the fields were set correctly
    - For the one-arg constructor that accepts a Scanner, do the same thing you did in the two-arg constructor to test the Scanner field, then check that the PrintStream field is equal to System.out.
    - For the one-arg constructor that accepts a PrintStream, do the same thing you did in the two-arg constructor to test the PrintStream field.
- Input/Output getters and setters
    - These tests should be pretty straightforward

    - They are effectively the same as your tests for fields in GelCap

Wrapping Up

- Fill out the retrospective template that you chose at the beginning of the project. If you think a different template would be better, you can change it first. Put the retrospective in the docs directory with the name Sprint1-Retro. You can use a word document, pdf, or simply create a text file and write each of the questions above your answers.
- Make sure that everything has been pushed to GitHub and that it has been merged into main.
- Your code should look clean and consistent, have adequate documentation (ie comments / JavaDocs), and pass checkstyle.
- Make a "ProductionAndVariations" branch when you are finished that branches off of main to save this iteration of your project. This is the branch that will be graded.
- Grade Breakdown

    60 pts: Code correctness (including tests)

    20 pts: Git Usage

    10 pts: Agile practices

    10 pts: Appearance / Checkstyle