Automated House Factory

CS 5004 Object Oriented Design

1. **Goals:**

* Create and logically use an enum
* Explore the use of abstraction
* Use inheritance correctly
* Use interfaces correctly to establish polymorphism
* Use switch statements logically
* Explore prototyping
* Study existing code

1. **In Recitation:**

This assignment uses a more complex design. In recitation I want you to explore the code and draw out the design for this application. Your TA will offer some tips and thoughts and be available for your questions. You’ll be submitting this design in UML or UML like format for an ICE for this week. Your recitation UML diagram will be included in your report. Your TA can approve your design before recitation is over.

1. **Instructions:**

One of the newest evolutions in games are idle games and resource management type games where you create resources that are then combined to form more complex items. We don’t have time to create one of these completely, but we could start one. What you’ll be creating this week is a set of factories. Each factory takes in a raw resource and produces blocks. These blocks can then be taken in by a more complex factory, a house factory, and generates a more complex structure. We’ll be reducing complexity a lot by ignoring some of the details. We are just building the prototype to test out our idea and theoretically help us design a more complete game later.

I’ll give you a head start by providing a driver I wrote to test my finished concept, but it’ll be up to you to implement the rest. You’ll know you were successful when you can run the driver correctly. Feel free to comment sections out and get it working one section at a time. Before you get started make sure to study the driver and understand what it is doing. Studying already created code is often part of the job.

Note: you can stop an infinite loop at the command line with CTRL + C.

We are also going to stray away from test driven design this time and think more in terms of prototyping. Prototyping is when we create code to help us with the design process and facilitate the creative process. In the end, we typically throw away the code we created and start with a more formal process like test driven development. However, you are welcome to create some JUnit tests for extensions if you would like and to get the practice.

Here’s a list of object descriptions and requirements. Use these, the rubric, and the already provided driver to create the finished application. There are always multiple ways to accomplish the same task, but try to alter the driver as little as possible.

Objective 1: Enum ResourceType

This object will define all the allowable types for your application. For this prototype use: stone, wood, and house.

Make sure to place the enum in global space so it is accessible from any file.

You should only create this variable once. Use a stand alone file to establish your enum.

Objective 2: Class Object Const

2.1 This class will hold constant variables you want to use throughout your application. That is its only purpose. Thus, make sure it can’t be inherited from.

2.2 All of its variables should be accessible from outside and unchangeable.

Inside this class, create constants for the weight of a stone block, the weight of a wood block, the number of stone blocks a house needs and the number of wood blocks a house needs.

Make sure to use the correct naming convention here.

Objective 3 : Class Object Resource

3.1 Think of this class object as raw material. You can have any amount including partial amounts. Thus, this will need to be a double. We’ll use weight to record this amount, but we’ll ignore what unit that weight is in. You will need a constructor for this Resource that takes a weight and a type.

3.2 A resource has a weight and a resource type of stone, wood, or house. Make sure the weight and type can’t be changed by other objects, but can be retrieved if needed.

3.3 You are also going to need a method to add to an amount of an existing resource of the same type. That just means you increase the weight of that resource. You also want to be able to subtract from a resource by reducing the weight.

3.4 Code defensively by throwing appropriate exceptions with appropriate error messages if another object tries to subtract more than what’s there already.

Objective 4 : Class Object Block

4.1 Block is a class that should never be instantiated. Make sure that isn’t possible.

4.2 It contains a resource type of stone, wood, or house and requires all children to create their own breakBlock method, but it doesn’t create a default one.

4.3 There are two constructors for a block.

One accepts a resource type and weight. This one can just be used to set the class instance variables.

The other just accepts two different types of resources. (The second is there so a house can be a block.) This should set the type to house and calculate the block weight based on the provided constants.

4.4 Make sure other class objects can get the type and weight of a block, but not set them. Go ahead and create an appropriate toString override as well.

Objective 5 : Wood and Stone blocks

5.1 Wood and Stone blocks are children of Block.

They each use a no argument constructor.

They make use of their parent's constructor that uses a weight and resource type within that no argument constructor and they set their weight based on the constants for a block of that type.

Make sure not to duplicate code here. Nothing that was passed down from their parent should be reused.

5.2 When a stone block is broken it returns a new resource containing all of the weight of the resource that was used to create it.

5.3 When a wood block is broken it only returns half that weight. In other words, stone can be reused, but wood can only be half reused.

Objective 6 : House blocks

6.1 A house block is a little different. Instead of a no argument constructor it takes stone and wood resources. Granted it would be better if we set it up to take blocks itself and that’s very possible, but we haven’t covered everything necessary for that yet.

6.2 When someone breaks a house the stone resource is returned, but not the wood.

6.3 Make sure to code defensively and throw appropriate exceptions if a class object tries to use the constructor without a sufficient amount of stone or wood resource. Make sure to use your constants to set this up.

Note: if we were to continue this implementation. These blocks would become even more different. Wood blocks may burn; stone blocks may take more effort to break; different blocks would have different images to represent them; etc.

Objective 7 : Factory interface

I’d like this application to be able to create many different factories in many different ways.

Thus, we need to establish a factory protocol: A factory should take a resource, produce a block, and display its inventory. How each factory does that will be up to that factory’s implementation.

Objective 8 : Stone and Wood block factories

8.1 A stone or wood block factory needs to have a resource bin(a resource object) to store resources and implement the factory interface. Once it is safe to do so, add any resource sent to it to the resource bin.

8.2 When it takes a resource, it should take that resource as an object. Later we might decide to send all kinds of things to a factory, but for now just throw an exception if we send it something that isn’t a resource we have already designed. You’ll need to explore how to use instanceof to get this done. Tip: don’t forget to type cast the object as a resource once you know it is safe to do so.

8.3 Go ahead and code defensively. Also throw an exception if an object tries to get it to take a resource that isn’t of the appropriate type. We are using exceptions here instead of something else because these are things that we aren’t going to leave up to a user. We actually want the program to stop if something like this occurs.

Tip: make sure to account for situations where a null object is sent to the factory

Objective 9 : House Factory

9.1 A house factory will produce houses instead of blocks. Instead of a resource bin, count the number of stone blocks and wood blocks it has in its inventory. It’ll have to check this number to see if it can produce a house yet.

9.2a Allow the take resource method to accept any object, but throw an exception if someone tries to send it something other than a block.

9.2b After it is safe to do so, typecast that object as a block and make sure it is a wood or stone block that was sent. Throw an exception if it isn’t one of those two types.

9.2c Now use a switch statement to add to the appropriate variable: If it is sent a stone block, add one to the stone block accumulator; if it is sent a wood block add one to the wood block accumulator.

9.3 When the factory is asked to produce a house, first check to see if it has enough of the appropriate type of blocks to do so. If it does, let it return a new house block; otherwise, just return null. Make sure to use the provided constants to get the amounts correct.

Objective 10 : Run the simulator

This time instead of a test file, I’m going to give you the completed driver class object. Studying this should help you complete the assignment. Reverse engineering existing code is a common part of the job. Often in industry we have to work backwards from legacy code and/or work with a team to finish an assignment.

<https://onlinegdb.com/tyFaf106A>

1. **Extensions:**

Outside academics you will not get specific requirements. Each lab assignment is worth 100 points, but the base requirements will only get you to 85% - 90%. If you want an A, you’ll have to find a way to go above and beyond what is asked. I’ll often make some suggestions to you in this section, but it is entirely up to you what you’d like to add to the assignment. Make sure you know who your grader is and discuss extension expectations with them. You won’t have to do all of the extensions to get 100% credit.

Extension suggestions:

* Add more than the requested exception handling
* Add additional functionality not requested
* Create a JUnit test file for at least some of your classes
* Add more types of resources and a more complete factory of your own
* Add useful comments to the provided driver
* Find a way to exercise the break block function
* Add a completely different driver that utilizes the classes in a different way

1. **Report:**

Each assignment must include a short report. The generation of this report should take you no more than 15 minutes. This gives you a chance to reflect back on what you learned and it makes grading easier on your grader. For this report, I want the following sections:

1. Reflection (*What did you learn?)*
2. How did this design incorporate future growth?
3. Did you change up the driver at all? If so, how?
4. Extensions (*What extensions are you requesting?)*
5. Your recitation UML diagram
6. Grading Statement (*Based on the rubric, what grade do you feel you deserve? Be honest.)*
7. **Submission:**

Please read carefully. Failure to follow submission instructions can result in a reduced score.

Submit all files on Canvas under the appropriate assignment. Make sure to include the following named as follows:

Submit your files as a single zip file named: “Your Name”\_”Assignment”.zip

Unless your grader requests it, do not submit your entire project folder. Do not include any JavaDocs, and make sure you remove any package statements. You may design this as a single file or as multiple. If you do it as a single file, it might be more difficult to add a JUnit file if you do so as an extension. If you submit multiple files make sure to submit the following:

* ResourceType.java
* Const.java
* Resource.java
* Block.java
* WoodBlock.java
* StoneBlock.java
* HouseBlock.java
* Factory.java
* StoneBlockFactory.java
* WoodBlockFactory.java
* HouseBlockFactory.java
* Main.java
* Report.pdf

Submission checklist:

* Did you include adequate comments?
* Did you include comment blocks at the top of each file?
* Did you name your files as requested?
* Does your code compile?
* Did you remove any package lines generated by your IDE?
* Did you take care of any warnings presented by your IDE?

|  |  |  |
| --- | --- | --- |
|  | **Possible** | **Given** |
| Objective 1 : Enum Resource Type | | |
| As requested | 3 | 0 |
| Objective 2 : Class Object Const | | |
| 2.1 | 3 | 0 |
| 2.2 | 3 | 0 |
| Objective 3 : Class Object Resource | | |
| 3.1 | 3 | 0 |
| 3.2 | 3 | 0 |
| 3.3 | 3 | 0 |
| 3.4 | 3 |  |
| Objective 4 : Class Object Block | | |
| 4.1 | 3 | 0 |
| 4.2 | 3 | 0 |
| 4.3 | 3 | 0 |
| 4.4 | 3 | 0 |
| Objective 5 : Wood and Stone Blocks | | |
| 5.1 | 3 | 0 |
| 5.2 | 3 | 0 |
| Objective 6 : House Blocks | | |
| 6.1 | 3 | 0 |
| 6.2 | 3 | 0 |
| 6.3 | 3 | 0 |
| Objective 7 : Factory Interface | | |
| implemented as requested | 3 | 0 |
| Objective 8 : Stone and Wood Block Factories | | |
| 8.1 | 3 | 0 |
| 8.2 | 3 | 0 |
| 8.3 | 3 | 0 |
| Objective 9 : House Factory | | |
| 9.1 | 3 | 0 |
| 9.2 | 3 | 0 |
| 9.3 | 3 | 0 |
| Objective 10 : Run the Simulator | | |
| Main works as requested | 5 | 0 |
| Misc | | |
| Report | 5 | 0 |
| Code Quality (correct indentation, comment blocks, variable naming, etc) | 10 | 0 |
| Not included in total possible: | | |
| Driver not included as requested | -100 | 0 |
| Does not compile | -100 | 0 |
| Extensions (Not calculated without report) | 10 | 0 |
| Late penalty | -20 | 0 |
| Creative or went above and beyond | 10 | 0 |
| Code contains warnings | -20 | 0 |
|  | |  |
| TOTAL POINTS POSSIBLE out of 100 | 89 | 0 |