**Exploring and Using Creational, Structural and Behavioural Design Patterns**

Matthew Sides

This report explores the various pattern types, creational, structural and behavioural patterns. It will define each pattern type, determining a design pattern from within each. It will go on to summarise and outline three specific design patterns that each belong to that type.

Furthermore, it will review what the pattern is capable of, along with a short summary and bullet pointed list of the commonality and usage of the pattern. Annotated and illustrated examples will be used, in addition to simplified pseudo-code which is also presented within this report to help better elaborate the points.

**Creational Pattern Type**

Creation design patterns in object orientated programming are design patterns that handle the creation of objects in a system, ensuring objects are created in a manner suitable to the situation. The majority of object orientated systems of any complexity require many objects to be instantiated over time, these patterns support the creation process by helping to provide the following capabilities:

* Generic instantiation - Allows objects to be created in a system without having to identify a specific class type within the code.
* Simplicity - Certain patterns simplify the object creation process making it easier, ensuring large complex code is not required to instantiate an object.
* Creation constraints - Some patterns enforce constraints on the type or number of objects that can be created within a system.

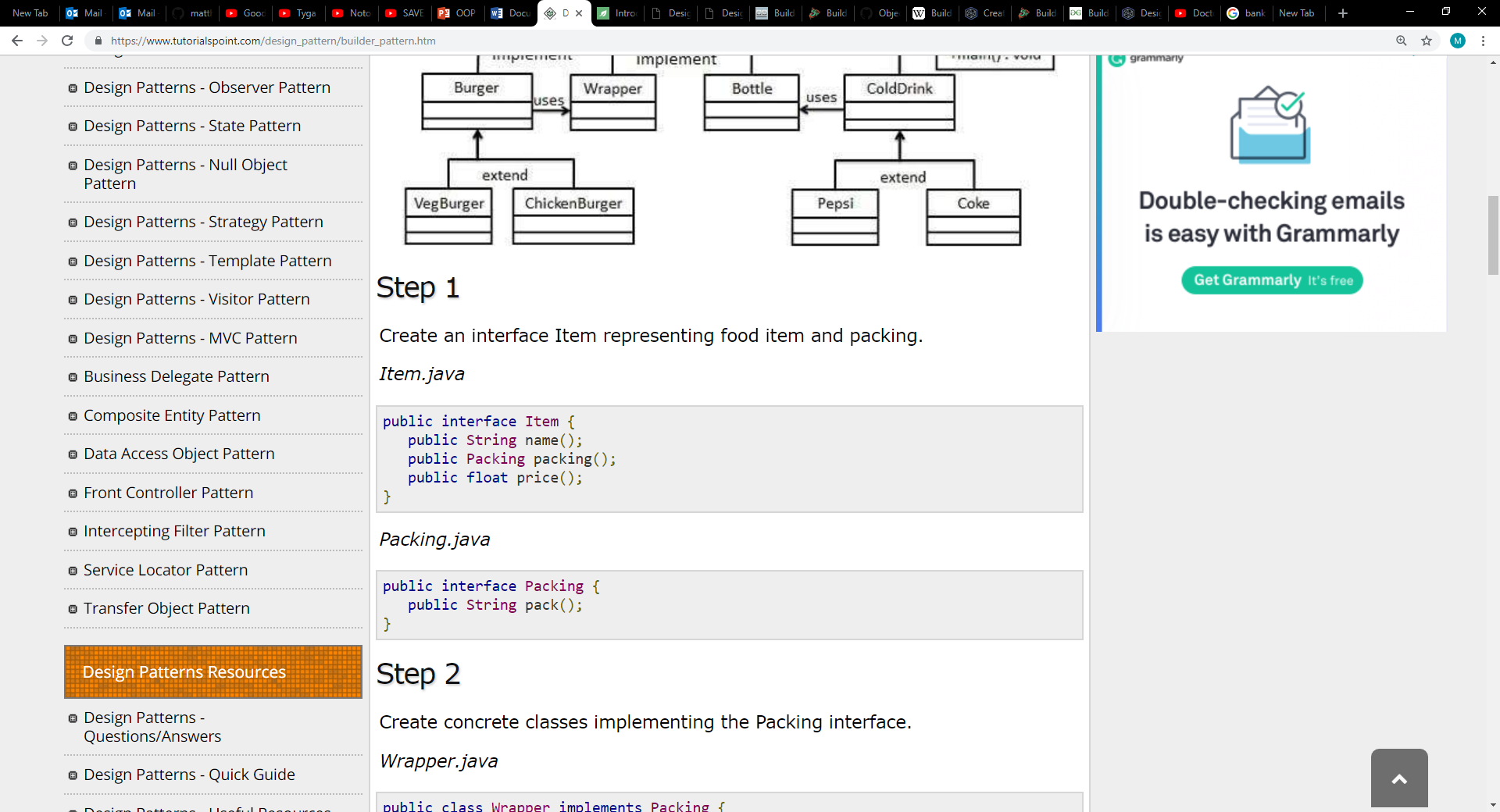
The following patterns detailed below are types of creational patterns:

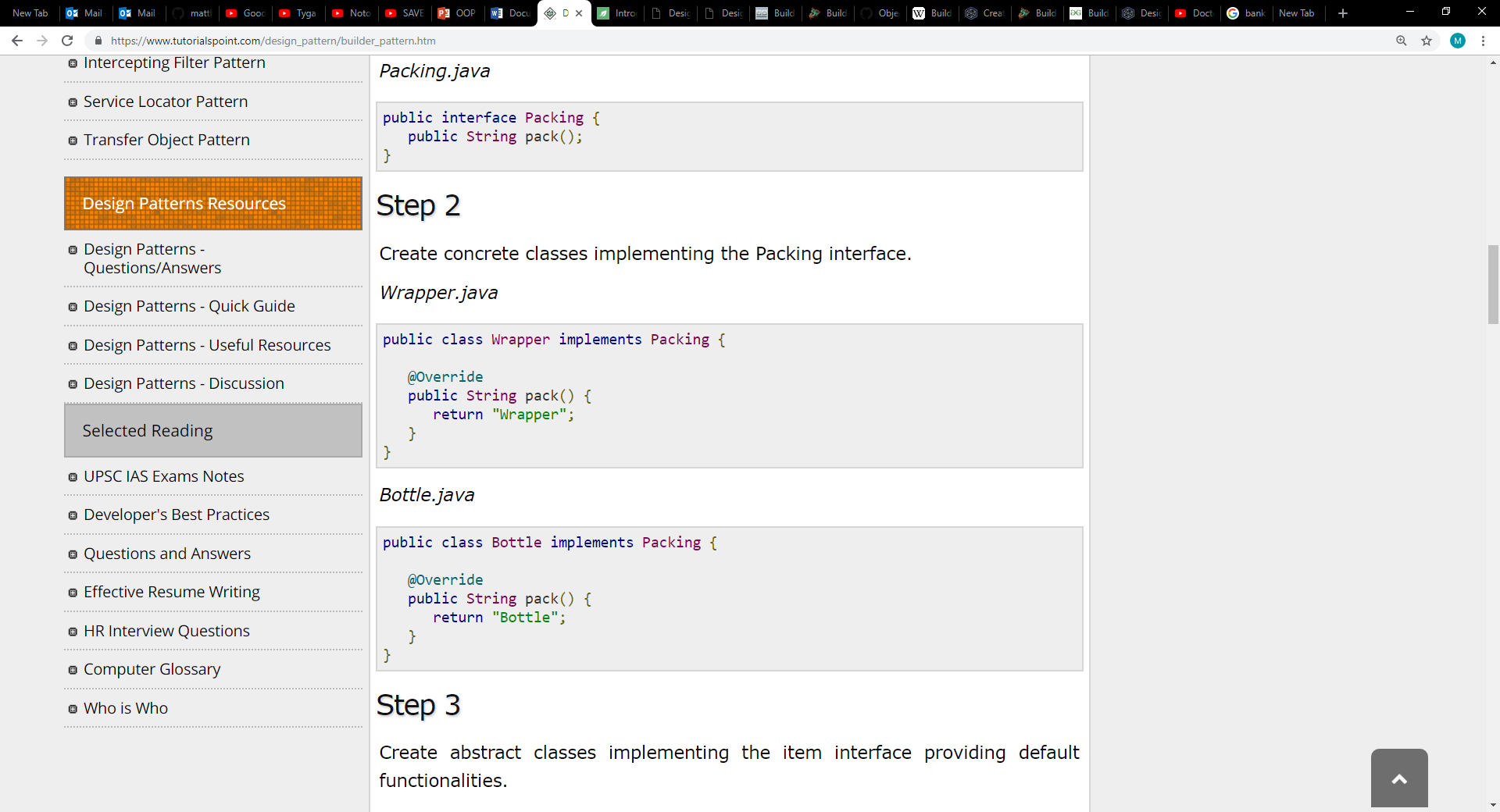
* Abstract Factory - is a creational design pattern that allows the production of a group (families) of related objects without specifying their concrete classes.
* Builder - is a collection of simple objects collated into a complex object, separating the construction and the representation so that multiple different representations can be created from the same process. In simpler terms the builder upon compiling builds the final object step by step (Tutorials Point, Design Pattern -Builder Pattern, no date).
* Singleton - is a class which restricts the instantiation of a class to one object, meaning that only a single instance can exist (Source Making, no date). This is practical when one object is required to coordinate actions across a system (Wikipedia, 2018).

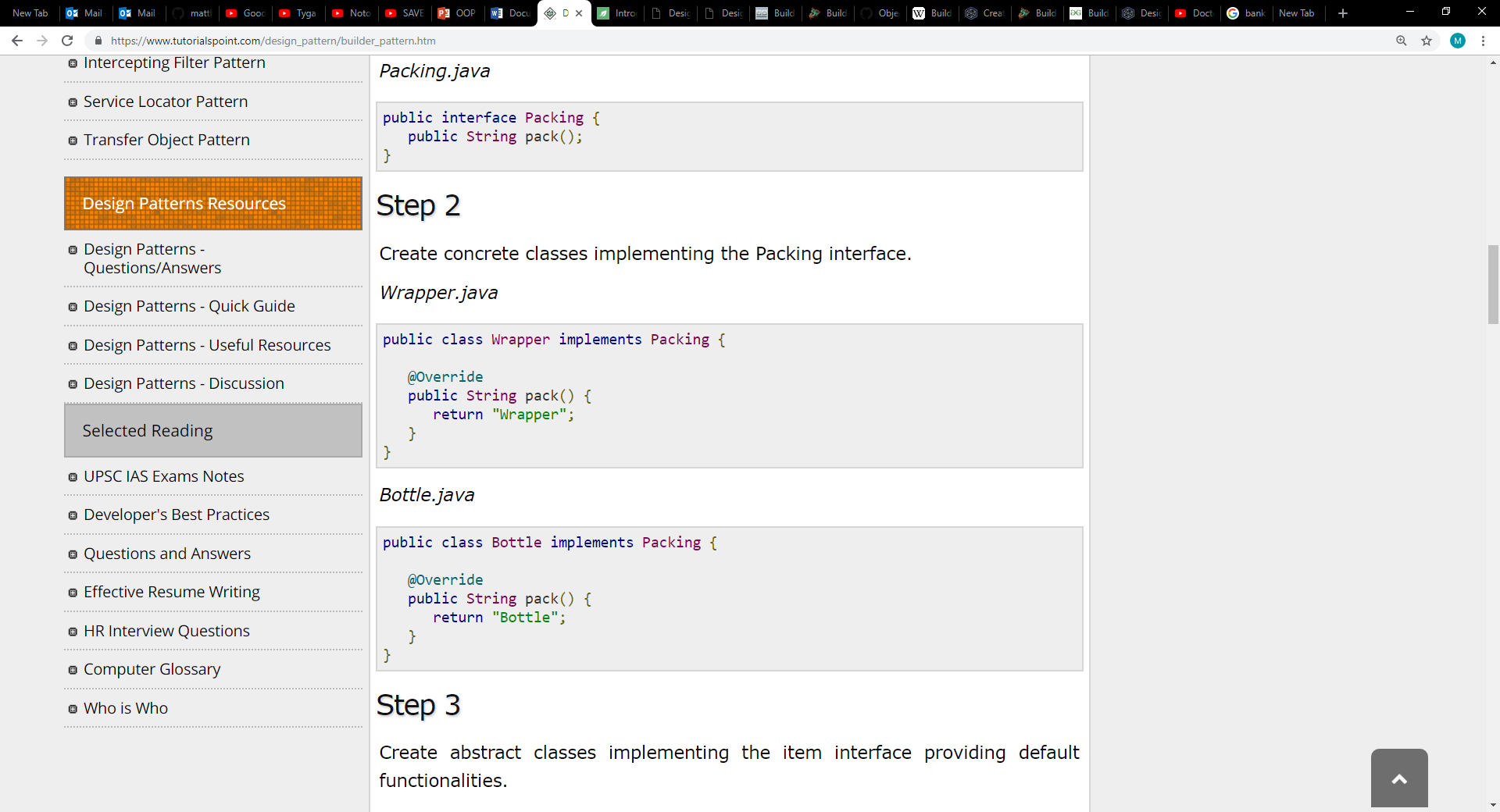
**Builder Pattern**

The ideology behind the Builder Pattern, as detailed by Gamma et al (1994. pg. 97), “Separate the construction of a complex object from its representation so that the same construction process can create different representations”. This refers to, as summarised above, the process of initialising a complex object that inherits simple objects, in which the outcome may alter based on the objects used upon construction. For instance, a meal is defined based on the ingredients used, below shows some code in which two meals are constructed upon compiling, one classified as a veggie meal and another a non-veggie meal.

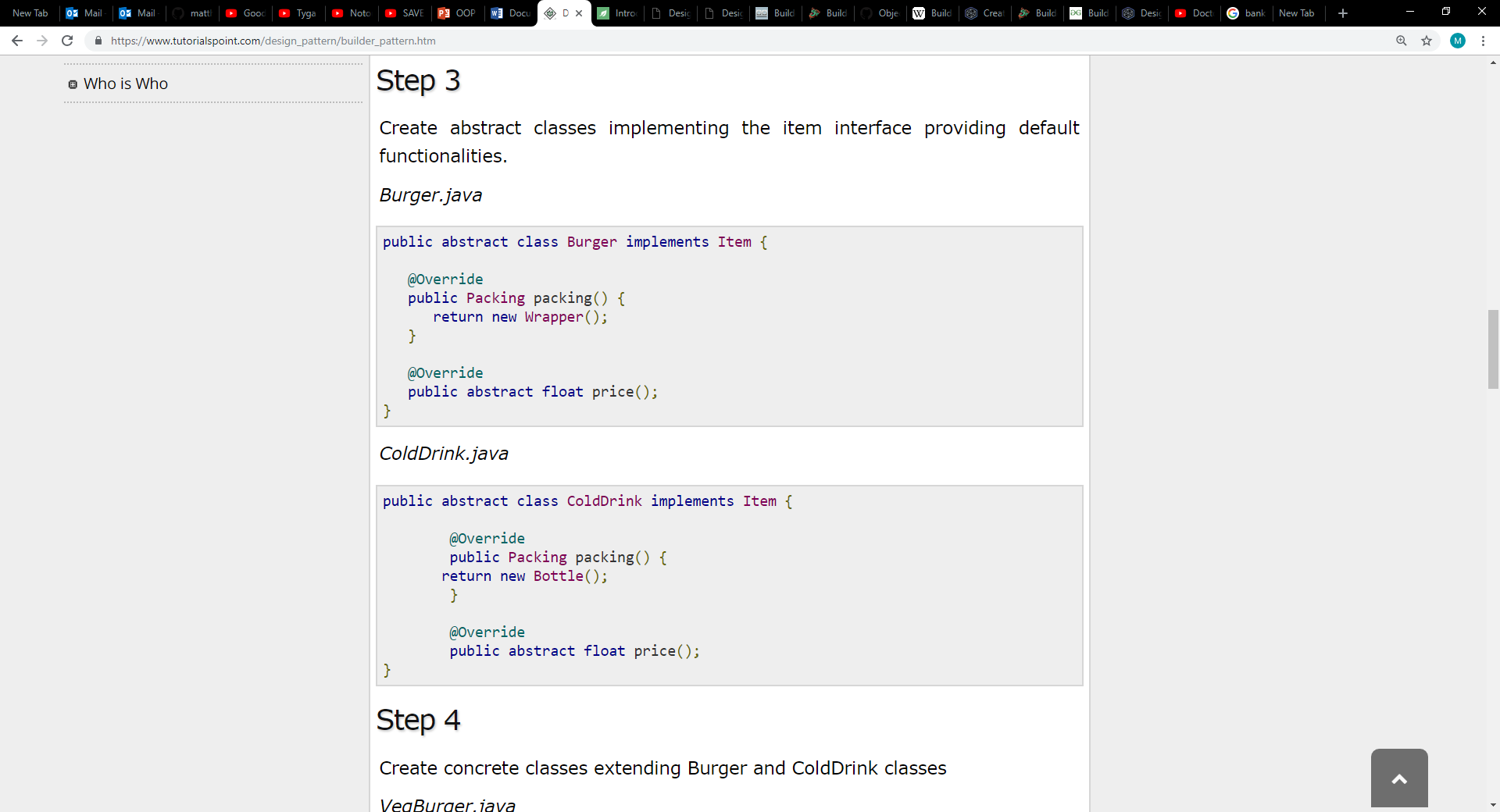
As shown below, essentially the attributes of an item along with the packaging are predefined, with the access modifier, type and variable name being set. The access modifier in this case is set to public, meaning that it is accessible to all classes. Concrete classes are then used to implement the packaging interface (essentially setting the string which is returned).







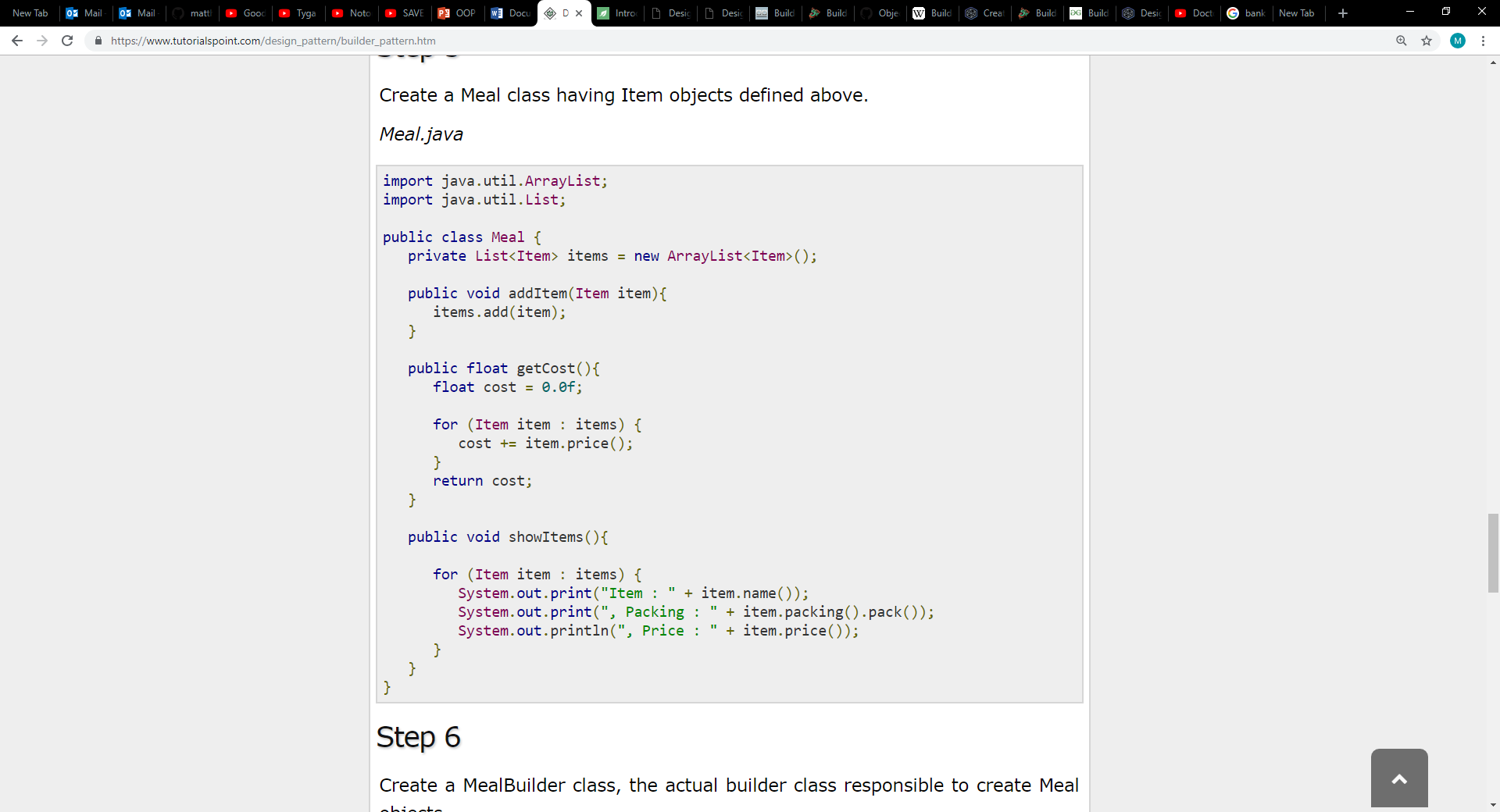
Abstract classes are also used in relation to the implementation of the item interface providing default functionalities i.e. defining the packaging of an item and price as shown below.



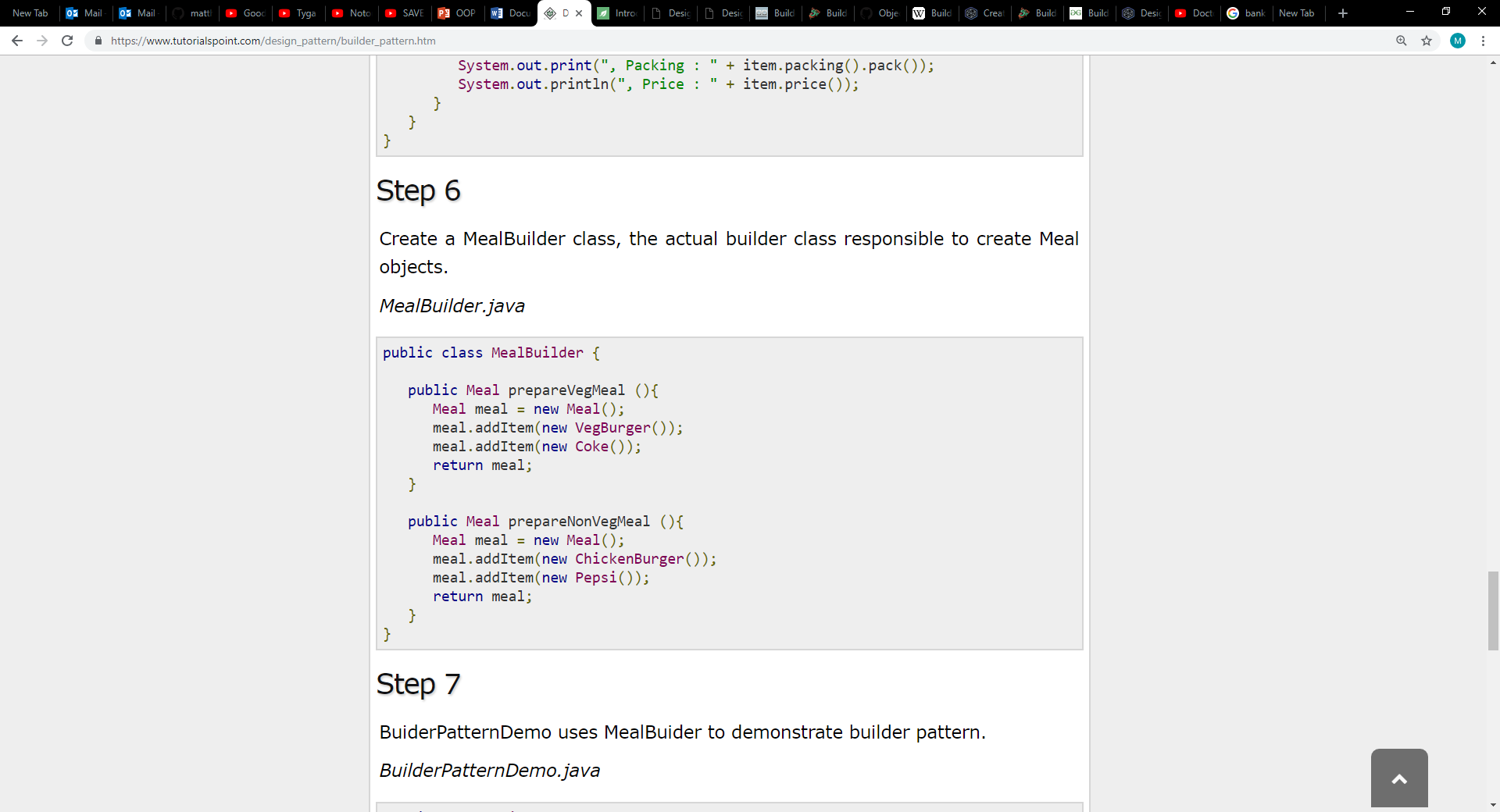
Whereas, subclasses, which are concrete are created and used to extend the abstract classes labelled ‘Burger’ and ‘Cold Drink’, setting a return value for the price and the types of burgers and cold drinks available (see code below).



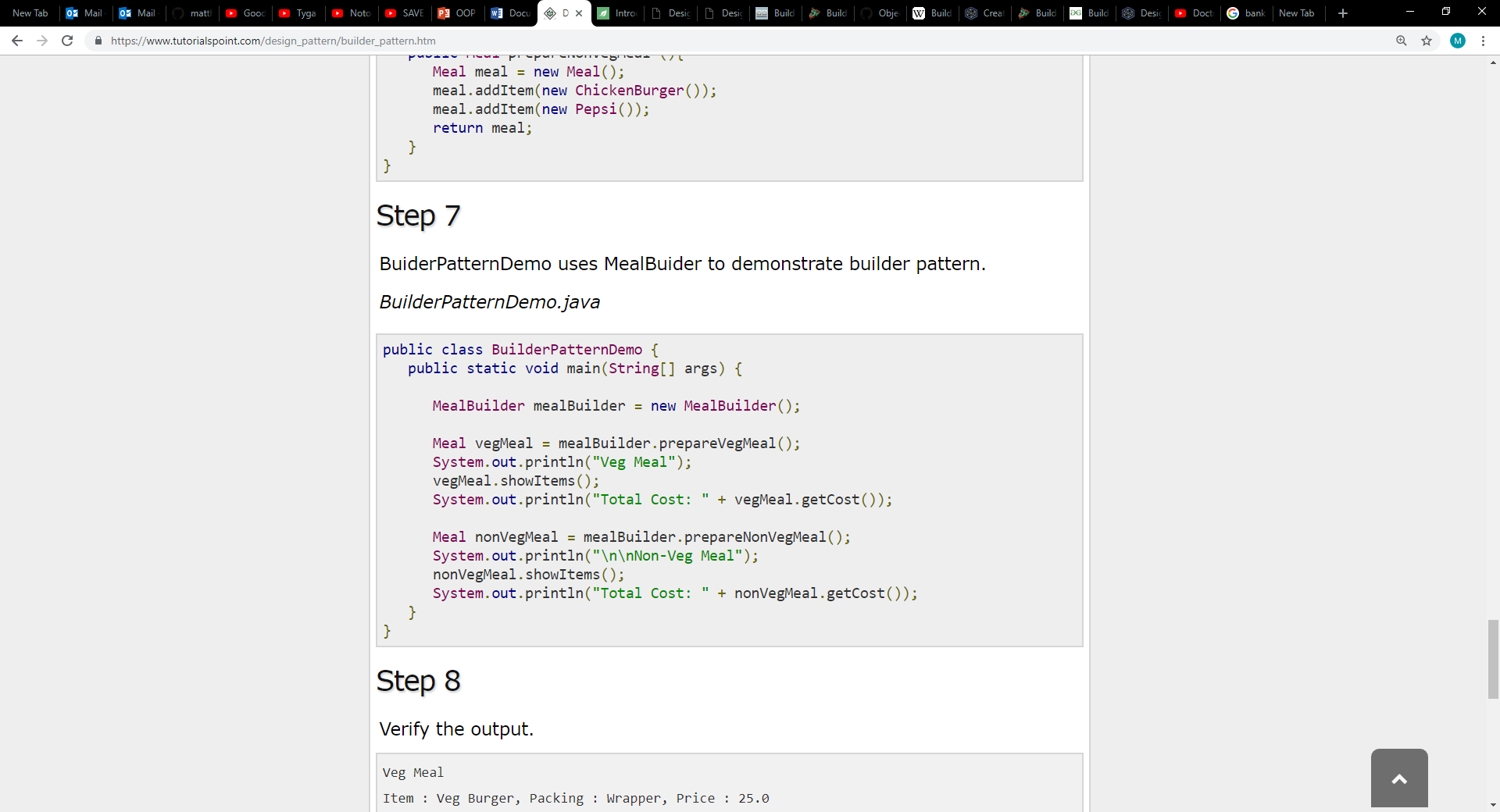
A meal class is then formulated, using the information (items) defined above to store into a list (array), which is then sorted through to calculate the price of said items, adding to the variable ‘cost’ which stores the overall price of the meal factoring in all the items in that specific meal class. Furthermore, printing the values of each item upon compiling onto the console using the ‘system.out.print’ command, as shown below.



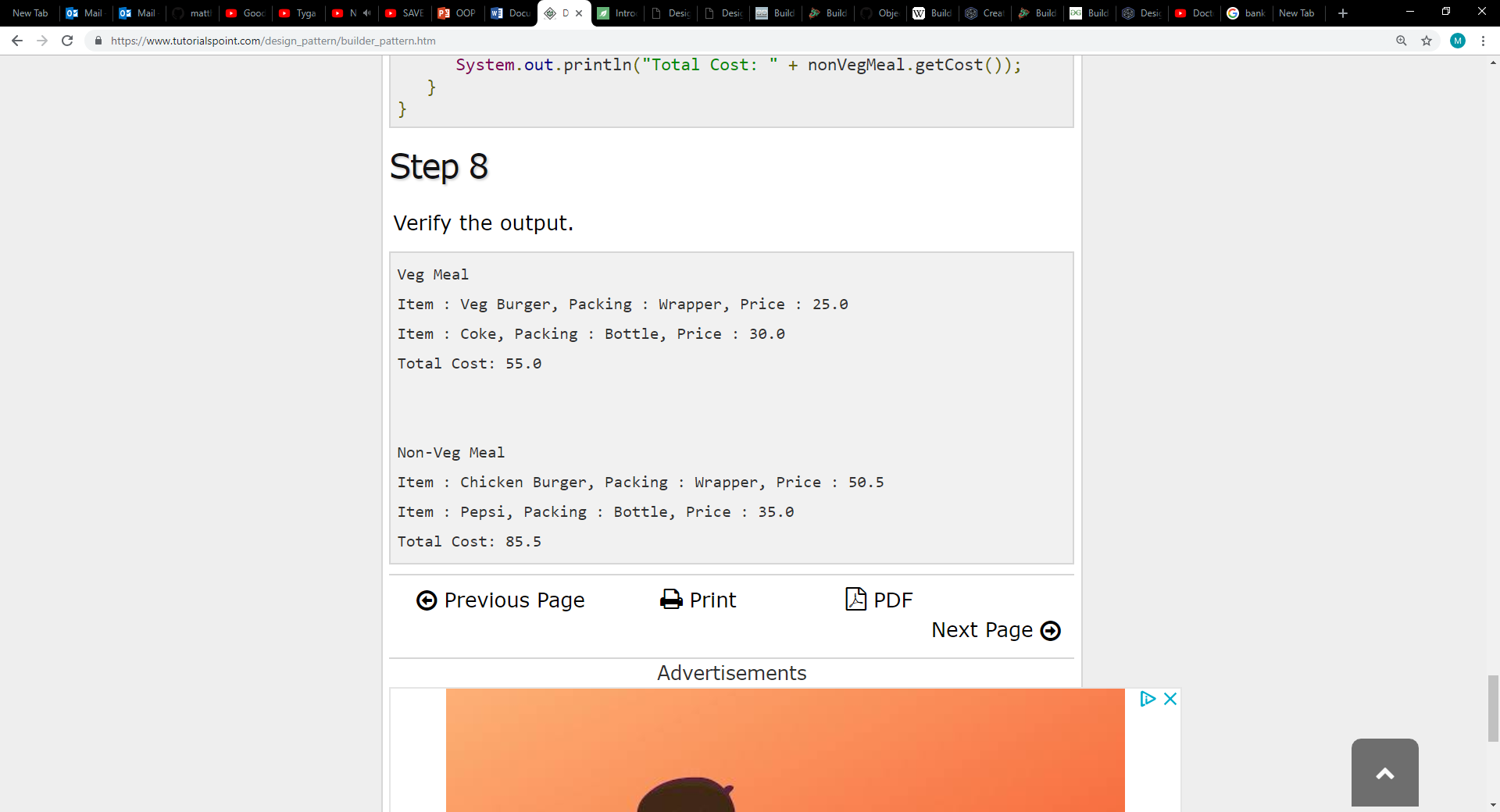
Thereafter a meal builder class, responsible for creating meal objects (complex objects), collates all the required objects to create the complex object, in this case known as either a Vegie meal or non-Vegie meal, returning the meal. See example below.

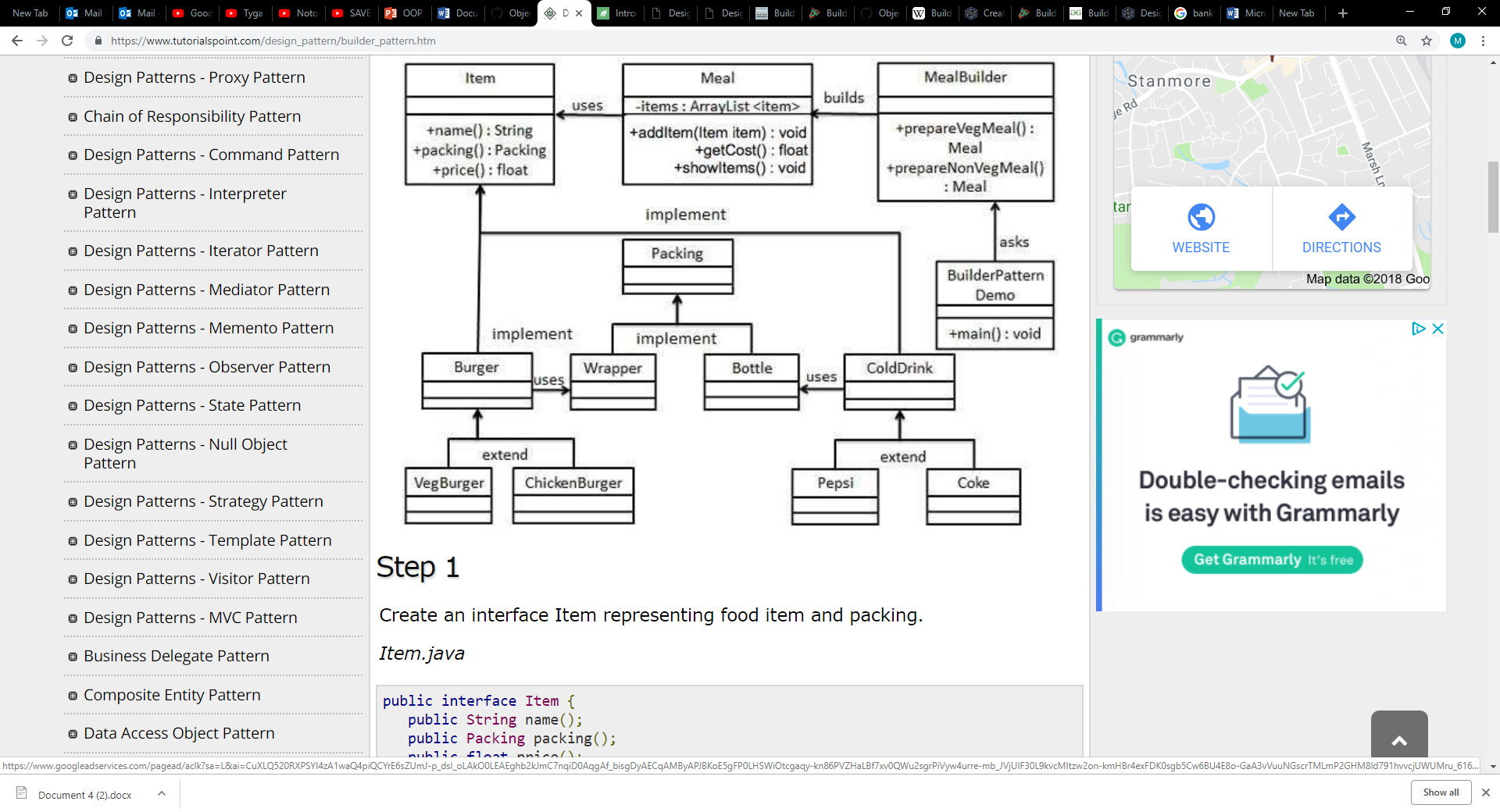


The Builder pattern is then demonstrated within the code snippet below using the meal builder to show the final product.



The final Output is displayed below, however the images were sampled from Tutorials Point, Design Pattern - Builder Patterns (No Date), certain images were excluded in order to effectively and efficiently illustrate this example (functions that function, however labelled differently etc.). A UML Diagram of the whole system can also be found below and on Tutorials Point as mentioned above.





The Builder Pattern should be used when either:

* The algorithm for creating a complex object is required to be independent of the parts that make up the object and how they're assembled.
* The construction process must allow different representations for the object that's constructed.

Builder pattern is particularly important as it allows for variation of a complex object's internal representation, providing an abstract interface, which allows the builder to hide the representation, internal structure and how the product is assembled. Furthermore, as the complex object is constructed through an abstract interface all that is required is for the internal complex objects representation to be changed (what is returned), defining a new kind of builder.

The builder pattern also isolates code for construction and representation, improving modularity by encapsulating the way a complex object is constructed and represented. Moreover, clients are not required to know about the classes that define the complex objects internal structure, not appearing in the output or interface. Each concrete builder contains all the code to create and assemble a specific complex object. The code is written once and can then be re-used to build different variants of the complex object from the same simple objects (parts).

Unlike creational patterns that construct complex objects in one shot upon compiling, the builder pattern constructs complex objects step by step, pairing this patterns step by step process along with the re-use of simple objects to create variants allows for finer control over the construction process and the details of the representation of the simple object.

**Review**

The fundamental basis of the builder pattern according to Gamma et al (1994, pg. 97), is to separate the construction and the representation so that multiple different representations can be created from the same process. In theory this is optimal as it allows users to segregate code into segments, refactoring each code snippet to generate different variants formed from the same parts. However, as the main class is formed through the reliance and merging of simple objects, if an object is changed it could see the class break. Although if an object has an aggregated relationship if one object is removed from the equation the other objects would still exist, though the built class as stated would break. What the pattern intends to achieve is to make it easier to create complex objects, which is achieved as the object's modularity allows for functionality change upon construction (compiling) without causing issues, which in turn allows for refactoring of code to create multiple variations formed from the same parts.

**Structural Pattern Type**

As note on Java T Point (no date), Structural Design Patterns are patterns concerned with how classes and objects can be composed, to form larger structures. Essentially, identify relationships to simplify the structure or design and focusing on how classes inherit and are composed from other classes.

The following patterns detailed below are types of structural design patterns:

* Adapter – functions like a bridge connecting two incompatible interfaces, combining the compatibility. This involves a single class those responsibility is to join functionalities of independent or incompatible interfaces (Tutorials Point, no date).
* Bridge - allows the separation of the abstraction from the implementation, abstraction referring to an interface or abstract class and the process of removing attributes (Geeks for Geeks, no date).
* Composite - Used when it is required to treat group objects in a similar way to a single object. Grouped objects are represented in a hierarchy, like a tree structure where each branch encases a function for it to perform (Tutorials Point, no date).

**Composite**

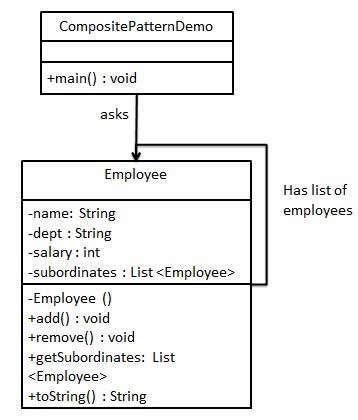
Composite design pattern is a pattern apart of the structural design pattern, which as noted by Reddy (2017), allows users to have a tree structure and ask each node within the tree to perform a task. This notion matches with the summary above and with Gamma et al (1994, Pg. 163), whom describe composite as “Compose objects into tree structure to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly".

As detailed on Geeks for Geeks - Composite Design Pattern (no date), the Composite Pattern has four participants:

* Component – Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
* Leaf – Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition. It also means that it cannot have objects or nodes below it.
* Composite – Composite stores child components and implements child related operations in the component interface. In comparison to leaf composite is the opposite and can have objects or nodes below it.
* Client – Client manipulates the objects in the composition through the component interface.

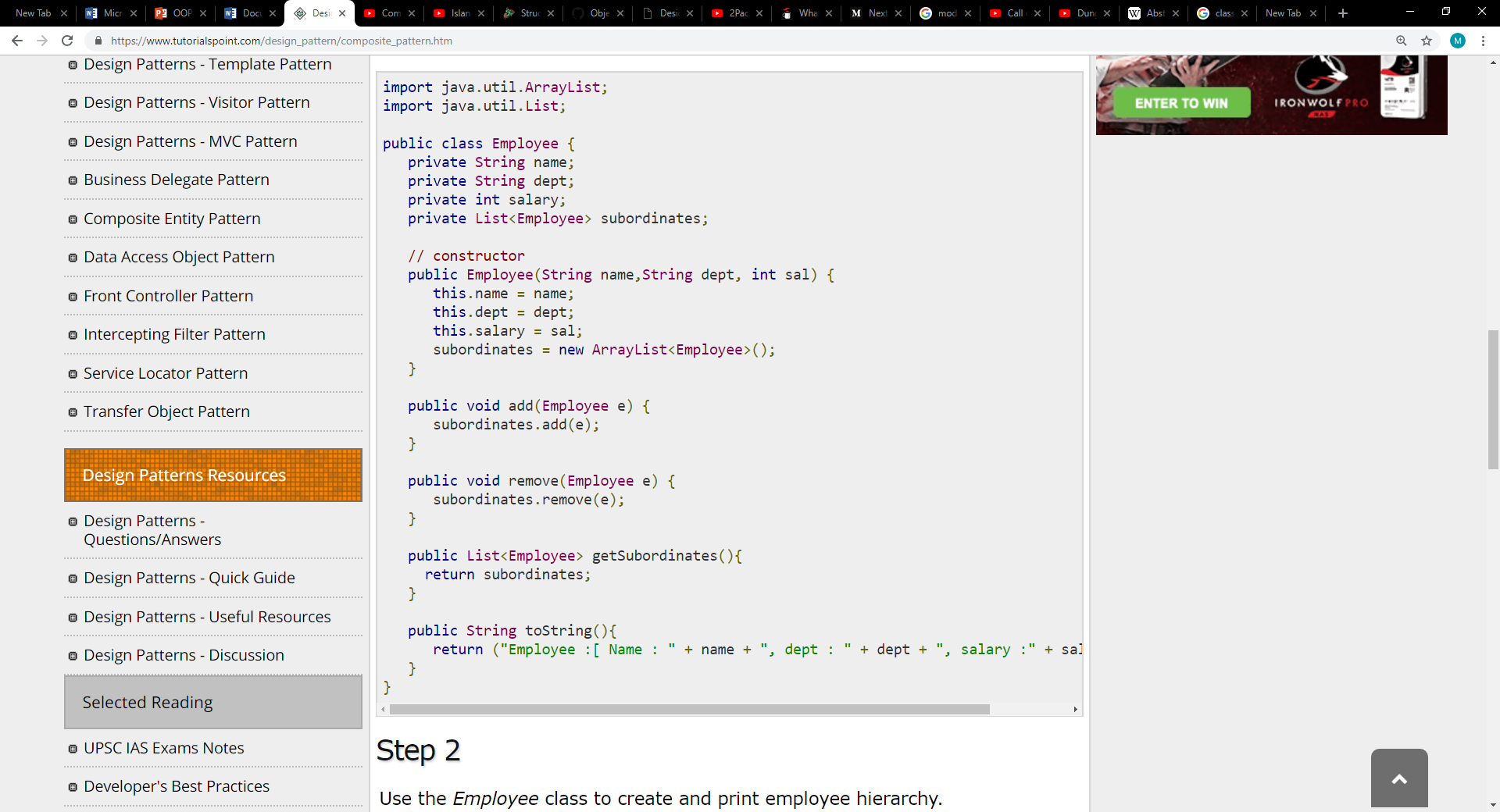
An example of composite is shown below in the form of code and a UML diagram:

The class ‘Employee’ acts as a composite pattern actor class. Whilst the ‘CompositePatternDemo’, utilises the Employee class to add department level hierarchy and print all employees.



UML Diagram (Tutorials Point - Composite Pattern, No Date)

The necessary libraries are first imported with the employee class being created and defined having a list of objects that have defined access modifiers (public, protected, private) ,return type or value and a labelled variable to store packets of information.



The employee class is then used to create and print the employee hierarchy, employing ‘for’ loops to continuously print out the employees until a condition is met, essentially getting all the Subordinates of the CEO and Subordinates of the head employee. In this segment of code certain jobs or objects are also added to the list and as stated printed using the ‘System.out.println’ function.



The final Output is displayed below, whilst all the images displayed in this example were sampled from Tutorial Point -Composite Pattern (No Date).



According to Gamma et al. (1994, Pg. 163), the composite pattern should be used when:

* It is required to represent part-whole hierarchies of objects.
* The user wants clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Patterns are important as they reduce the number of objects in turn reducing the amount of memory used, eliminating or lessoning the possibility of memory related errors. Moreover, through sharing objects the execution time of said program can be lessened.

**Review**

The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Through doing this client can perform operations on an object without needing to know the number of objects inside. Although treating heterogeneous (diverse) collection of objects transparently, meaning without the user being aware, requires the sub-object interface to be defined at the root of the composite class hierarchy (abstract component class). However, doing this runs the risk of clients attempting to do futile actions, for instance adding or removing objects from leaf objects. Though if the sub object interface is defined in the composite class for security reasons, transparency is lost as the leaves and composites now use different interfaces. Essentially the composite pattern in theory and practice works composing objects into trees representing part whole hierarchies, however what is gathered is that composite although not forcing the user to treat all components as composites, the user must put all operations that they want to be treated uniformly in the component class else it will not work in the way the user may intend. Whilst operations that are not required to be treated uniformly should not be included.

**Behavioral Pattern Type**

The following patterns detailed below are types of Behavioural design patterns:

* Command - is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event later. This information includes subsets of the following: an object, a method to be applied to the object, and the arguments to be passed to the invoker upon request. The invoker object then searches for the appropriate object which can handle this command, passing it to the corresponding object which then executes the command (Wikipedia: Command Pattern, 2018).
* Mediator – a behavioral design pattern used to facilitate and reduce communication complexity between multiple objects or classes. The pattern restricts direct communications between objects, opting to use a mediator class forcing collaboration, also providing easy maintenance through loose coupling (ensuring code modularity and that components depend on each other to the least extent practicable) (Refactoring Guru, no date).
* State – is a behavioral design pattern which functions similarly to a finite state machine, allowing an object to alter its behavior when it’s internal state changes. Although it is important to note this is not a software implementation of a finite state machine (Wikipedia: State Pattern, 2018).

**Command**

The Command pattern as summarised above, is a behavioral design pattern in which a request is wrapped under an object as a command and passed to invoker object. The invoker object then searches for the appropriate object which can deal with this command and passes the command to the corresponding object, which then executes the command.

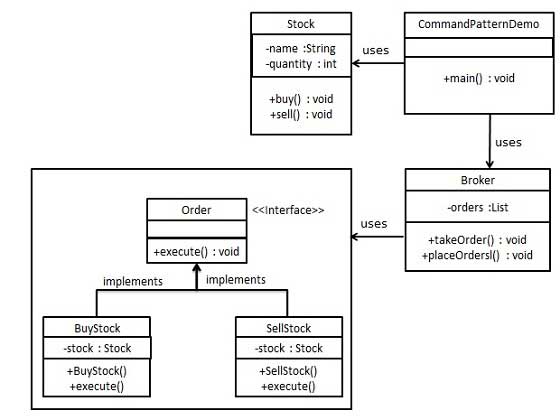
The Command Pattern has 5 participants (classes) (OODesign.com, no date):

* Command - declares an interface for executing an operation;
* Concrete Command - extends the Command interface, implementing the Execute method by invoking the corresponding operations on Receiver. It defines a link between the Receiver and the action.
* Client - creates a Concrete Command object and sets its receiver;
* Invoker - asks the command to carry out the request;
* Receiver - knows how to perform the operations; [9]

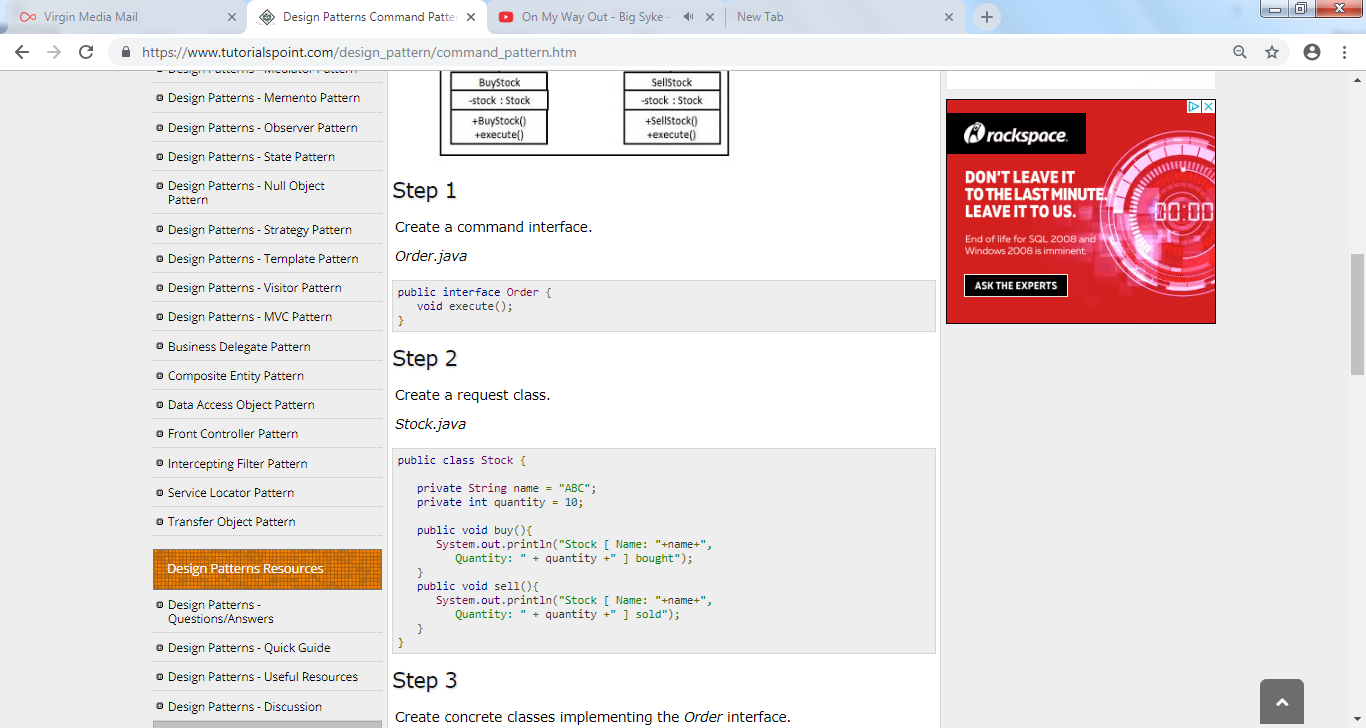
An example of the command pattern is shown below in the form of code and a UML diagram:

Interface Order acts as a command, Whilst Stock class acts as a request. The concrete command classes BuyStock and SellStock implements the Order interface which takes care of the actual command processing. A class Broker acts as an invoker object that can take and place orders.

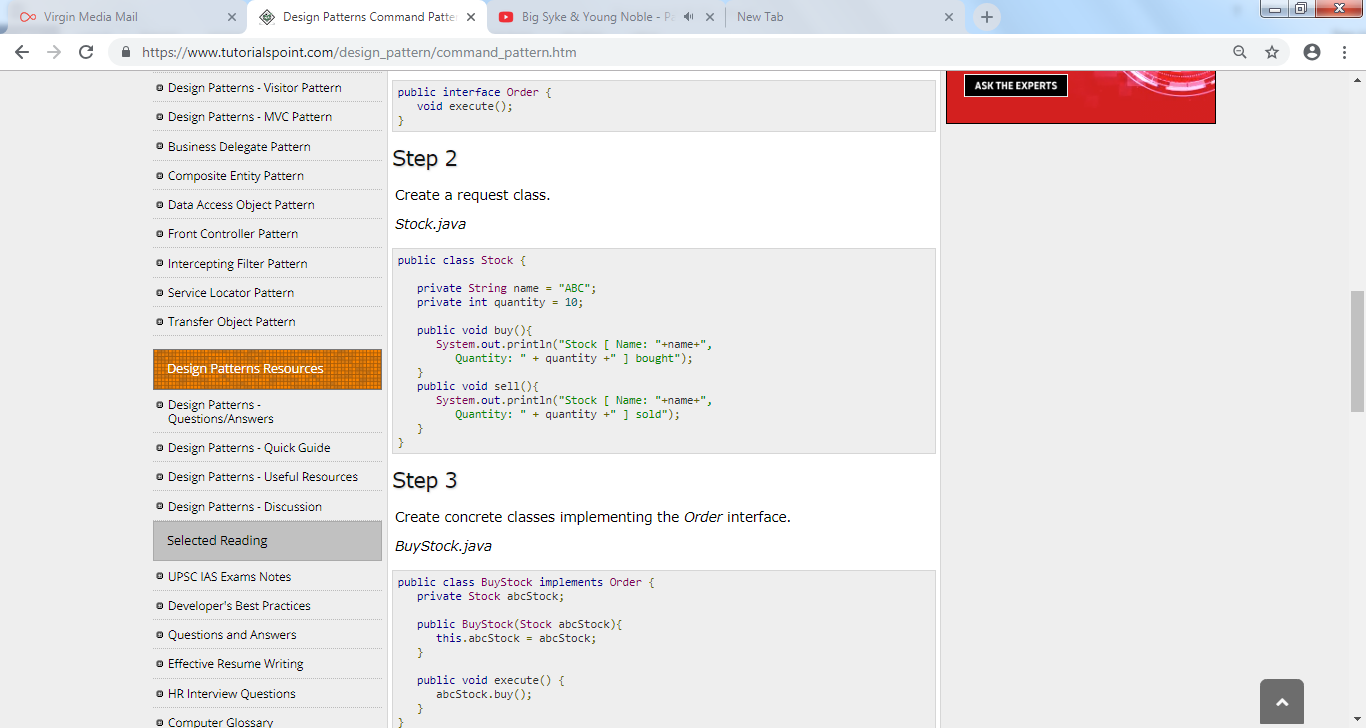
The Broker Object uses the command pattern to identify which object will execute which command based on the type of command.



A command interface is created with the access modifiers being defined along with the variable execute.

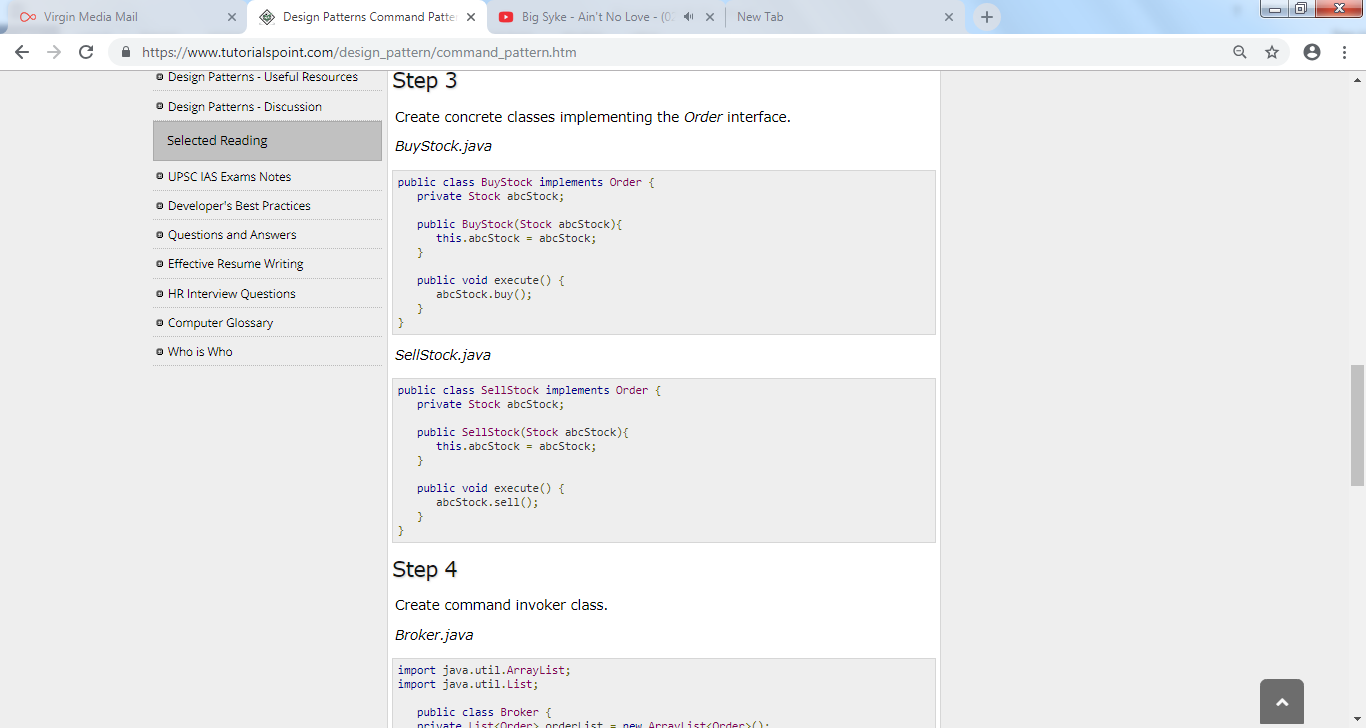


Request classes are defined and created, printing the quantity of stocks bought and sold.

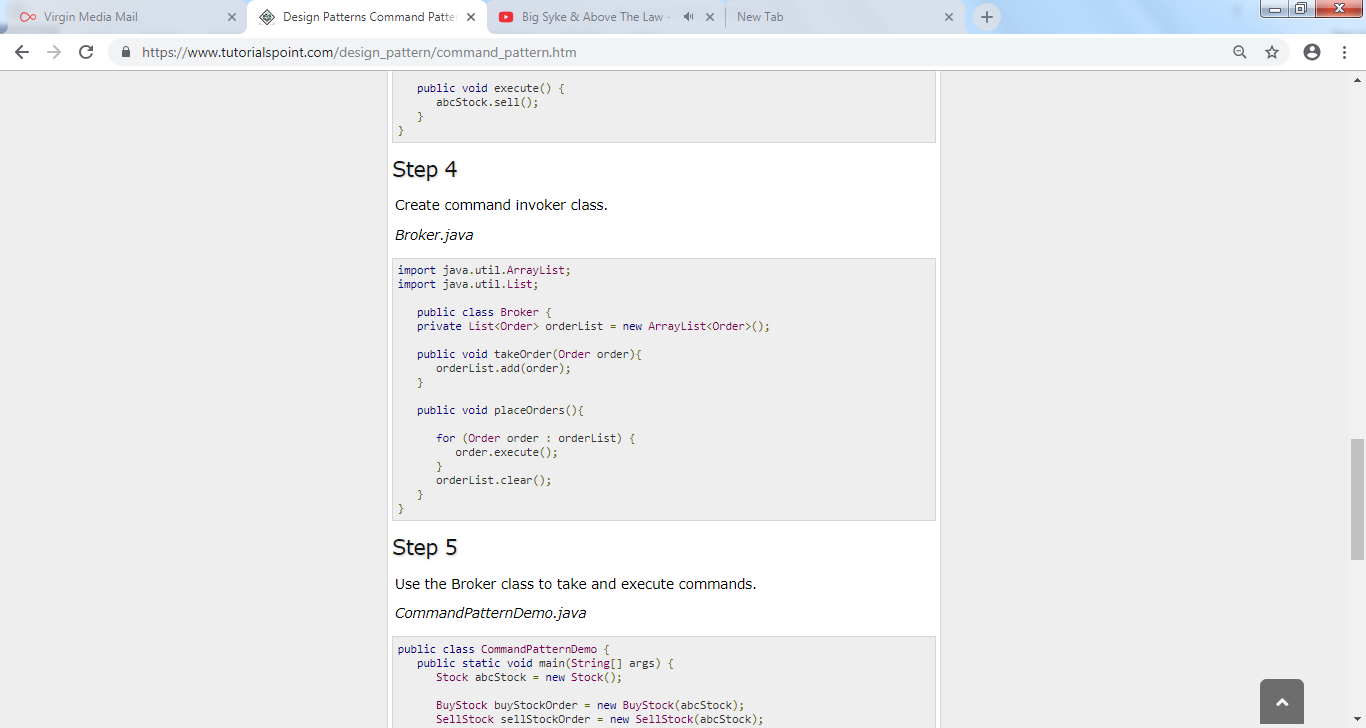


Concrete Classes are created implementing the order interface.





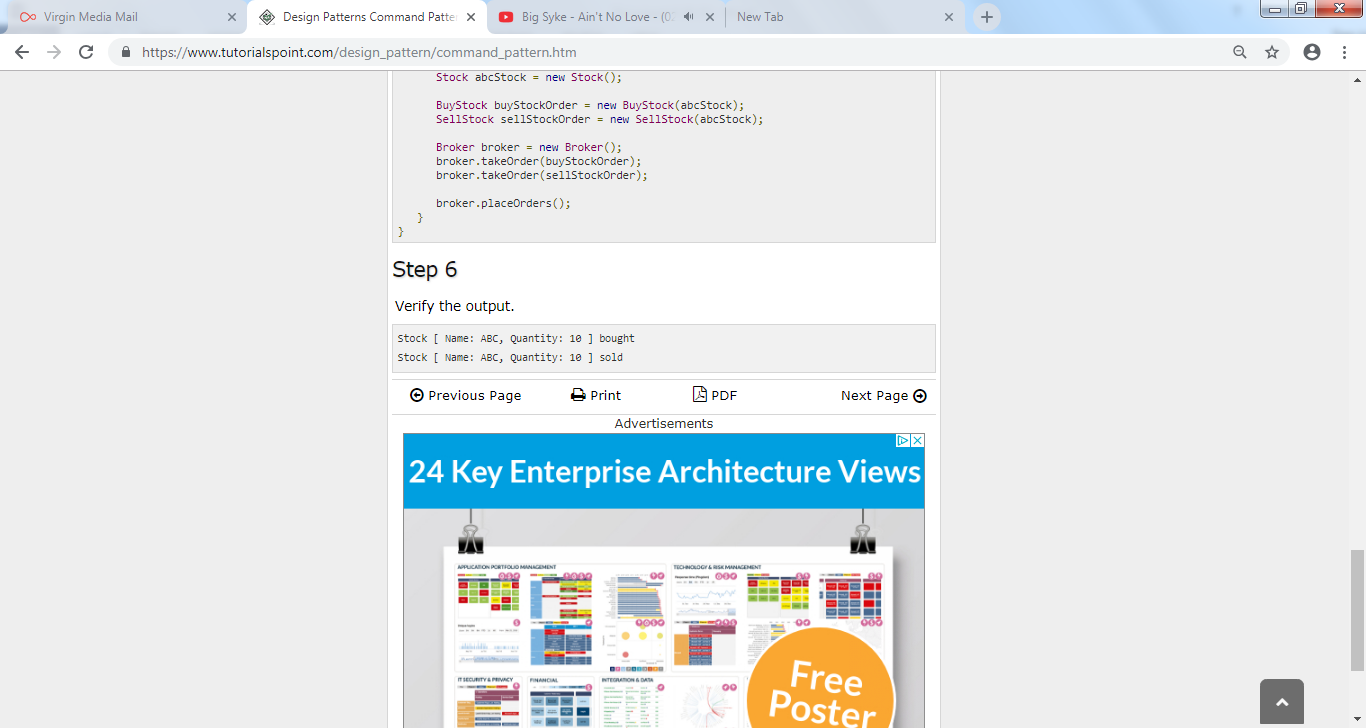
The invoker class is created which asks the commands to carry out the request.



The Broker class is used to take and execute commands.



The final Output is displayed below, whilst all the images displayed in this example were sampled from tutorialspoint (No Date).



As noted by Teo (2012), the command pattern should be used when:

* Users want to be able to parameterize objects by an action to perform.
* The queue requires specification and to execute requests at different times.
* When a set of changes to data need to be encapsulated as a single action (i.e. a transaction).

The command pattern is important as the executor of the command does not require information detailing what the command is, what context information it needs on or what it does. All of the information is encapsulated in the command. This essentially makes it plausible to have a list of commands that are executed in order that are dependent on other items, assigned to triggering an event etc.

**Review**

The command pattern intent as summarised above, is to encapsulate everything required to take an action and allows the execution of the action to occur completely independently of any of that context. If this is not a requirement then the pattern is not applicable to the situation. Moreover, based on the information collated, the command pattern is optimal fulfilling its purpose in addition to allowing the creation of a sequence of commands through a queue system, extensions that make it easy to add a new command without altering existing code. Also decoupling (decoupling refers to minimising the reliance, dependencies and relationships between objects and classes, noted by Giard (2009)) the classes that invoke the operation from the object that knows how to execute the operation.

**Bibliography**

Gamma, Erich., Helm, Richard., Johnson, Ralph., Vlissides, John. (1994). Design Patterns. Pg 97. Addison-Wesley: United States.

Geeks for Geeks. No date). Bridge Design Pattern. Available at: <https://www.geeksforgeeks.org/bridge-design-pattern/> {Accessed: 04/12/2018]

Geeks for Geeks. (No date). Composite Design Pattern. Available at: <https://www.geeksforgeeks.org/composite-design-pattern/> [Accessed: 05/12/2018]

Giard, David. (2009). Intro to OOP. Available at: <http://davidgiard.com/2009/08/06/IntroToOOPPart3OOPConcepts.aspx> [Accessed: 07/12/2018]

Java T Point. (No date). Structural Design Patterns. Available at: <https://www.javatpoint.com/structural-design-patterns> [Accessed: 04/12/2018]

OODesign.com. (No date). Available at: <https://www.oodesign.com/command-pattern.html> [Accessed: 07/12/2018]

Refactoring. (No date). Mediator. Available at: <https://refactoring.guru/design-patterns/mediator> [ Accessed: 06/12/2018]

Source Making. (No date).Creational Patterns. Available at: <https://sourcemaking.com/design_patterns/creational_patterns%20> [Accessed: 03/12/2018]

Teo, Ignatius. (2012). Understanding the Command Design Pattern. Available at: <https://www.sitepoint.com/understanding-the-command-design-pattern/> [Accessed: 07/12/2018]

Tutorials Point. (No date). Design Patterns and Adaptor Patterns. Available at: [https://www.tutorialspoint.com/design\_pattern/adapter\_pattern.htm](https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm%20) [Accessed : 04/12/2018]

Tutorials Point. (No date). Design Pattern - Builder Pattern. Available at: [https://www.tutorialspoint.com/design\_pattern/builder\_pattern.htm](https://www.tutorialspoint.com/design_pattern/builder_pattern.htm%20%20)  [Accessed on: 03/12/2018]

Tutorials Point. (No date). Design Pattern - Command Pattern. Available at: <https://www.tutorialspoint.com/design_pattern/command_pattern.htm> [Accessed: 07/12/2018]

Tutorials Point. (No date). Design Patterns - Composite Pattern. Available at: <https://www.tutorialspoint.com/design_pattern/composite_pattern.htm> [Accessed: 04/12/2018]

Wikipedia. (2018). Command Pattern. Available at: [https://en.wikipedia.org/wiki/Command\_pattern](https://en.wikipedia.org/wiki/Command_pattern%20) [Accessed: 06/12/2018]

Wikipedia. (2018). Singleton Pattern. Available at: <https://en.wikipedia.org/wiki/Singleton_pattern> [Accessed: 03/12/2018]

Wikipedia. (2018). State Pattern. Available at: <https://en.wikipedia.org/wiki/State_pattern> [Accessed: 06/12/2018]

Reddy, Navin. (2017). Composite Design Patten Theory. Available at: <https://www.youtube.com/watch?v=mp5lwolO-wM> [Accessed: 05/12/2018]