Workbook

Design Patterns
(SE – 404)

Name	
Roll No	
Batch	
Year	
Department	

Workbook

Design Patterns
(SE – 404)

Prepared by

Dr. Saman Hina
Assistant Professor, CS&IT

Approved by

Chairman
Department of Computer Science & Information Technology

Table of Contents

Lab # 1

Object:

Reviewing the functional decomposition and structured programming.

Theory:

Functional decomposition is a common method of software development. In this method, a complex problem can be broken into simple steps which are easy to understand. In other words, functional decomposition is a natural way to deal with complexity.

For instance, if you want to write a code to access a description of shapes that were stored in a database and then display them. The solution might contain following steps:

1. Locate the list of shapes in the database.
2. Open up the list of shapes.
3. Sort the list according to some rules.
4. Display the individual shapes on the monitor.

   Step # 4 can further be broken down into following steps.

   4a. Identify the type of shape.

   4b. Get the location of the shape.

   4c. Call the appropriate function that will display the shape, giving its shape location.

Problem with functional decomposition:

- One main program with all responsibilities.
- Sequence and calling of functions.
- Do not facilitate changes.

Structured Programming is an alternate way to functional decomposition approach. This can be called modular programming. Structured programming provides set of modules that follows a logical structure in the program to make it more efficient and easier to understand and modify.

Exercise:

1. Form a group of 3-5 students and write a module on step 4c “Call the appropriate function that will display the shape, giving its shape location.” Use functional decomposition approach.

2. Use structured programming approach to solve the following example.

Suppose that you are an instructor at a conference. People in your class have another class to attend following yours, but don’t know where it is located. One of your responsibilities is to make sure everyone knows how to get to the next class.

Lab # 2

Object:

Reviewing the concepts of object oriented software development.

Theory:

The Object Oriented Paradigm

The object-oriented paradigm is centered on the concept of the object. Everything is focused on objects. Objects have traditionally been defined as data with methods (the object-oriented term for functions). The advantage of using objects is that I can define things that are responsible for themselves. Objects inherently know what type they are. The data in an object allows it to know what state it is in. and the code in the object allows it to function properly (that is, do what it is supposed to do).

The best way to think about an object is to think of something with responsibilities. A good design rule is that objects should be responsible for themselves and should have those responsibilities clearly defined. Remember that objects have data to tell the object about itself and methods to implement required functionality. Many methods of an object will be identified as callable by other objects. The collection of these methods is called the **object's public interface**.

Consider the classroom example of previous lab; you could write the **Student** object with the method **gotoNextClassroom()**. There is no need to pass any parameters because each **Student** object would be responsible for itself. That is, it would know:

- What it needs to be able to move
- How to get any additional information it needs to perform this task

Initially, there was only one kind of student – a regular student who goes from one class to class. Note that there would be many of these "regular students" in the classroom (system). There should have an object of each student, to track the state of each student easily and independently of other students. However it is insufficient to require each Student object to have its own set of methods to tell it what it can do and how to do it, especially for the tasks that are common to all students.

A more efficient approach would be to have a set of methods associated with all students that each one could use or tailor to his or her own needs. 'General student' can be defined that contain definition of all these common methods. Then it can have all manner of specialized student, each of whom has to keep track of his or her own private information. In object oriented terms, this general student is called a *class*. A class is a definition of the behavior of an object. It contains a complete description of the following:

- The data elements the object contains.
- The methods the object can do.
- The way these data elements and methods can be accessed.

To get an object, we have to tell the program that we want a new object of this type. This new object is called an instance of the class. Creating instances of a class is called *instantiation*.

Exercise:

In the example explained in this lab, there is an assumption that any type of student is allowed into the collection (either regular student or graduate student). The problem is how do we manage the collection to refer to its constituents? In coding, this collection will actually be an array or something, of some type of object. If the collection were named something like **RegularStudent**, we will not be able to put graduate students into the collection. If collection is just a group of objects, how can it assures that wrong type of object is not included (that is, something that doesn't do "Go to your next class")?

Work in group of 3-5 students to find a solution to this problem.

Lab # 3

Object:

The UML – Unified Modeling Language.

Theory:

The UML – Unified Modeling Language is a visual language (meaning a drawing notation with semantics) used to create models of programs. The UML include several diagrams for each process area. Some diagrams are for analysis, some for design and others for implementation and deployment. Each diagram shows the relationship among the different sets of entities, depending on the purpose of the diagram.

The UML is useful for communication among the team members about what is required to be done. The UML gives us tools to understand better requirements rather than moving to development phase straight away. In this context, we will cover two diagrams;

1. The Class diagram
2. The Sequence diagram

The Class Diagram

This is the most basic and commonly used UML diagram. It not only describes classes but also shows the relationships among them. Different types of relationships possible are;

- When one class is a "kind of" another class; is-a relationship.
- When there are associations between two classes;
  - ✓ One class "contains" another class: has-a relationship (Aggregation)
  - ✓ One class "uses" another class: the uses-a relationship (Dependency)
  - ✓ One class "creates" another class (Composition)

The Sequence Diagram

Sequence diagrams shows how objects interact with each other. These are most common type of interaction diagrams. Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. The diagrams are read left to right and descending.

Exercise:

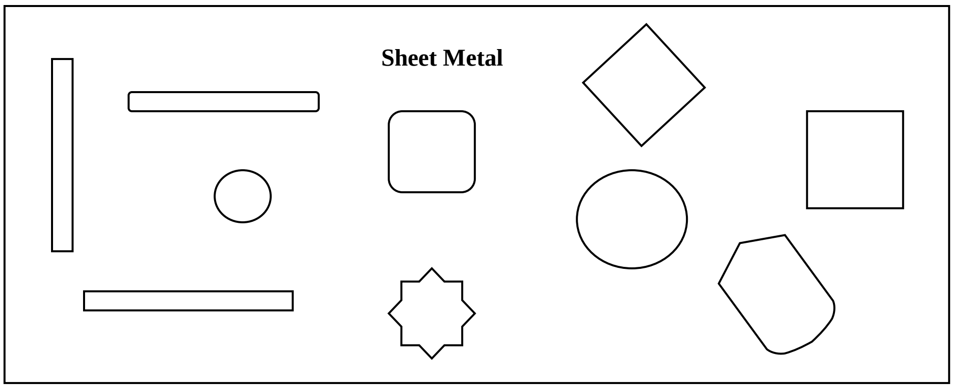
Draw a class diagram and sequence diagram for a computer tool that need to extract information from the two different versions of the CAD/CAM system. Both versions are not compatible with each other. This information extraction is required so that an expert system could use it in a particular way. The expert system needed this information to control the manufacturing of the part. Because the expert system was difficult to modify and would have a longer lifespan than the current version of the CAD/CAM system, this information extracting tool should be written in a way that it could easily be adapted to new revisions of the CAD/CAM system. The vocabulary of this system is described in Table 1 and Table 2.

Table 1 CAD/CAM Terminology

Term	**Description**
Geometry	The description of how a piece of sheet metal looks: the location of each of the features and their dimensions and the external shape of the sheet metal.
Part	The piece of sheet metal itself and the geometry of each part is to be stored.
Dataset or Model	The set of records in the CAD/CAM database that stores the geometry of a part.
NC machine and NC set	Numerically controlled (NC) machine. A special manufacturing tool that cuts metal using a variety of cutting heads that are controlled by a computer program. Usually the computer program is fed the geometry of the part. This computer program is composed of commands called the NC set.

Table 2 Features found in a Piece of sheet metal

Shape	Description
Slot	Straight cuts in the metal of constant width that terminate with either squared or rounded edges. Slots may be oriented to any angle.
Hole	Circles cut into the sheet metal. Typically they are cut with drill bits of varying width.
Cutout	Squares with either squared or rounded edges. These are cut by a high-powered punch hitting the metal with great impact.
Special	Performed shapes that are not slots, holes, or cutouts.
Irregular	Anything else. They are formed by using a combination of tools.



Lab # 4

Object:

Introduction to Design Patterns and understanding the Facade Pattern.

Theory:

Design patterns are part of the cutting edge of object-oriented technology. Object-oriented analysis tools, books, and seminars are incorporating design patterns. The most commonly stated reasons for studying design patterns are because patterns enable us to;

- **Reuse Solutions –** By reusing already established designs, get a head start on your problems and avoid designing solution from scratch. You should get the benefits from other's experiences and do not have to reinvent solutions. Experienced designers reuse solutions that have worked in the past.
- **Establish Common Terminology –** Communication and teamwork require a common base vocabulary and a common view point of the problem. Design patterns provide a common point of reference during the analysis and design phase of a project.

The Gang of four[1] suggests a few strategies for creating good object-oriented designs. In particular, they suggest the following:

1. Design to interfaces.
2. Favor aggregation over inheritance.
3. Find what varies and encapsulate it.

The Façade Pattern

According to the Gang of Four, the intent of the Façade pattern is to:

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Key Features

Intent- You want to simplify how to use an existing system. You need to define your own interface.

Problem- You need to use only a subset of a complex system. Or you need to interact with the system in a particular way.

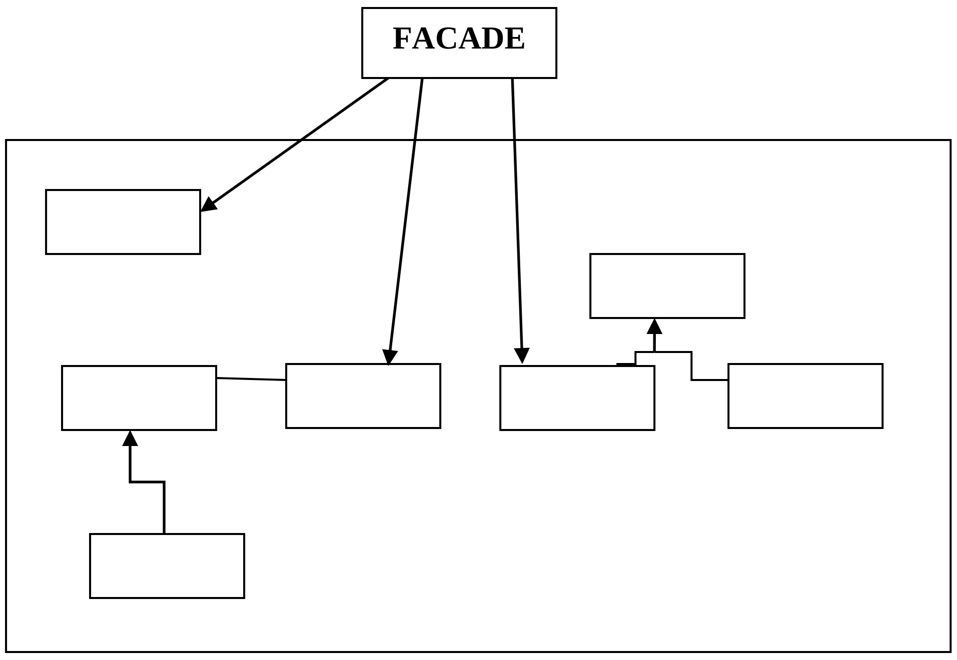
[1] Gamma, Helm, Johnson and Vlissides (Writers of *Design Patterns: Elements of Reusable Object-Oriented Software*) .

Solution- the Façade pattern presents a new interface for the client for the existing system to use.

Participants and Collaborators- It presents a simplified interface to the client that makes it easier to use.

Consequences- the Façade simplify the use of the required subsystem. However, because the Façade is not complete, certain functionality may be unavailable to the client.

Implementation- Define a new class (or classes), that has the required interface. Have this new class use the existing system.



Exercise:

Work in group of three people. Consider any example of a system that illustrates the need of Façade. Explain key features of Façade pattern with respect to your example system and also draw diagram. Justify the use of Façade pattern in your example.

Lab # 5

Object:

Understanding the Adapter Pattern.

Theory:

The Adapter Pattern

Convert the interface of a class into another interface that the clients expect. Adapter let classes work together that could not otherwise because of incompatible interfaces.

Intent- Match an existing object beyond your control to a particular interface.

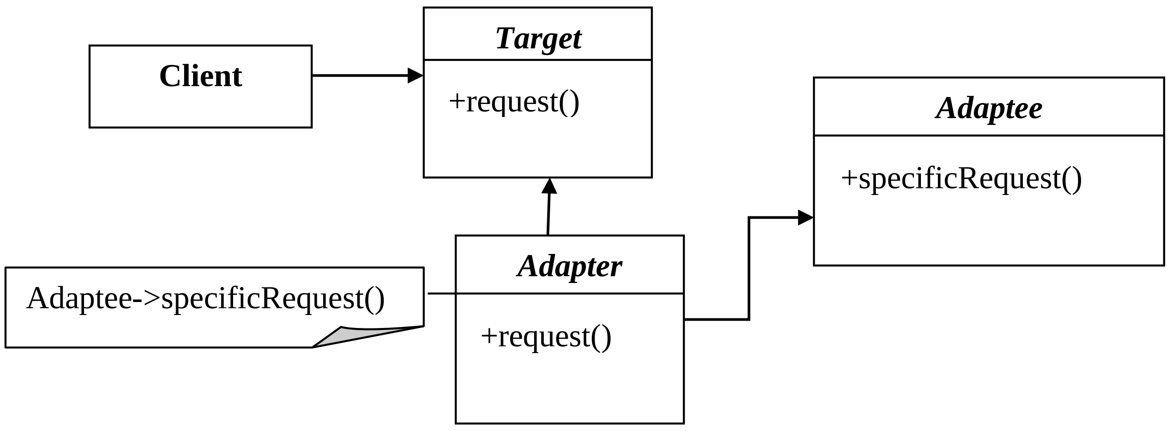
Problem- A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.

Solution- The Adapter provides a wrapper with the desired interface.

Participants and Collaborators – The Adapter adapts the interface of an Adaptee to match that of the Adapter's Target (the class it derives from). This allows the Client to use the Adaptee as if it were a type of Target.

Consequences- The Adapter Pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.

Implementation- Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.



Exercise:

Work in group of three people. Consider any example of a system that illustrates the need of Adapter Pattern. Explain key features of Adapter pattern with respect to your example system and also draw diagram. Justify the use of Adapter pattern in your example.

Lab # 6

Object:

Understanding the Strategy Pattern.

Theory:

The Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Intent- Enables you to use different business rules or algorithms depending on the context in which they occur.

Problem- The selection of an algorithm that needs to be applied depends on the client making the request or the data being acted on. If you just have a rule in place that does not change, you do not need a strategy pattern.

Solution- Separate the selection of an algorithm from the implementation of the algorithm. Allows for the selection to be made based upon context.

Participants and Collaborators

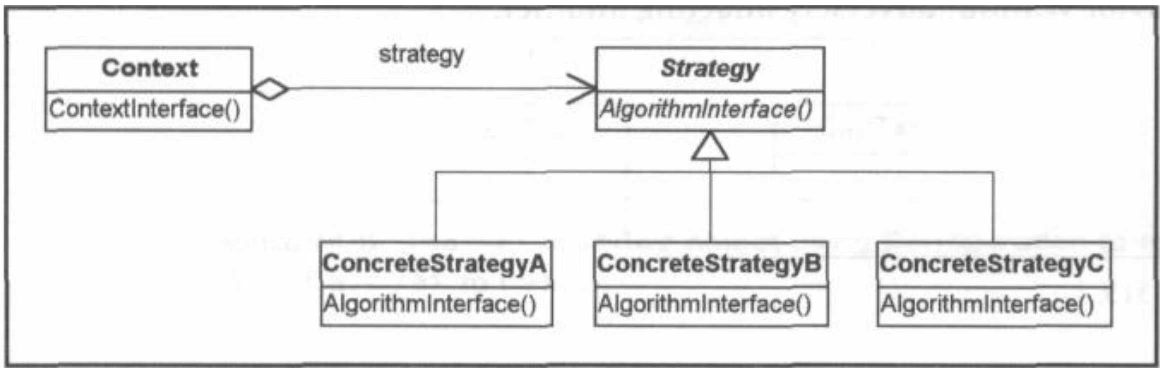
- **Strategy** specifies how the different algorithms are used.
- **ConcereteStrategies** implement these different algorithms.
- **Context** uses a specific **ConcreteStrategy** with a reference of type **Strategy. Strategy and Context** interact to implement the chosen algorithm. (Sometimes Strategy must query Context.) The **Context** forwards requests from its client to **Strategy**.

Consequences-

- The Strategy pattern defines a family of algorithms.
- Switches and/or conditionals can be eliminated.
- You must invoke all algorithms in the same way. (They must all have the same interface.) The interaction between the **ConcreteStrategies** and the **Context** may require the addition of methods that get state to the **Context.**

Implementation- Have the class that uses the algorithm (Context) contain an abstract class (Strategy) that has an abstract method specifying how to call the algorithm. Each derived class implements the algorithm as needed.

Note: This method wouldn’t be abstract if you wanted to have some default behavior.



Exercise:

Work in group of three people. Consider a simple Shopping Cart where users have options to pay for their purchases in two different ways – using Credit Card or using PayPal. Elaborate the implementation of Strategy pattern with respect to this example system and also draw diagram. Justify using the key features of Strategy pattern.

Lab # 7

Object:

Understanding the Bridge Pattern.

Theory:

The Bridge Pattern

According to the Gang of four, the intent of the Bridge pattern is to;

“Decouple an abstraction from its implementation so that the two can vary independently”

Intent- Decouple a set of implementations from the set of objects using them.

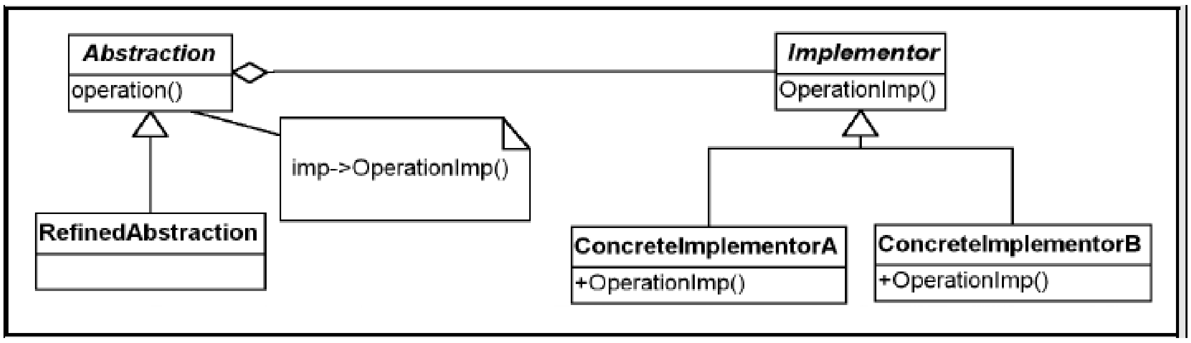
Problem- The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes.

Solution- Define an interface for all implementations to use and have the derivations of the abstract class use that.

Participants and Collaborators- Abstraction defines the interface for the objects being implemented. **Implementor** defines the interface for the specific implementation classes. Classes derived from **Abstraction** use classes derived from **Implementor** without knowing which particular **ConcreteImplementor** is in use.

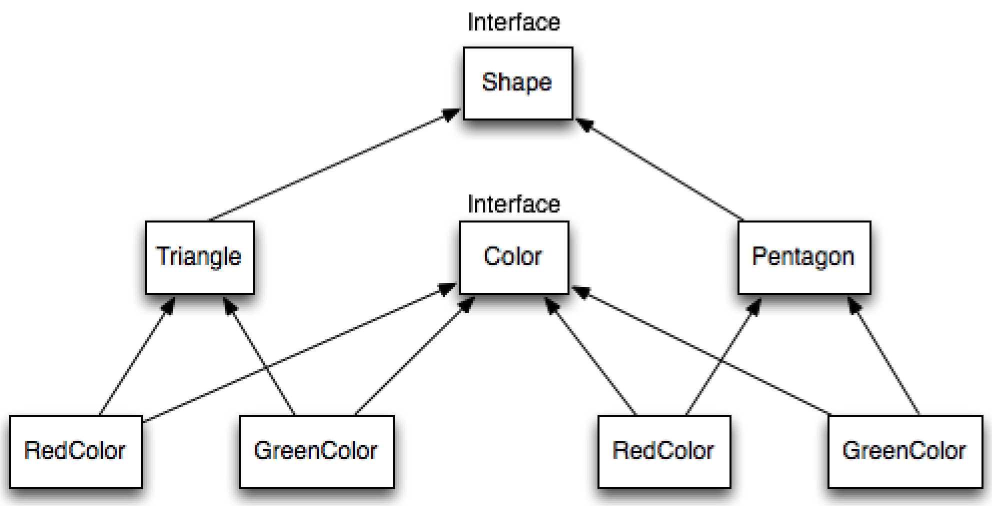
Consequences- The decoupling of the implementations from the objects that use them increases extensibility. Client objects are not aware of implementation issues.

Implementation- 1) Encapsulate the implementations in an abstract class. 2) Contain a handle to it in the base class of the abstraction being implemented.



Exercise:

Work in group of three people and apply bridge pattern to the following scenrio. Also justify the usage of bridge pattern with the help of its key features.



Lab # 8

Object:

Understanding the Abstract Factory pattern.

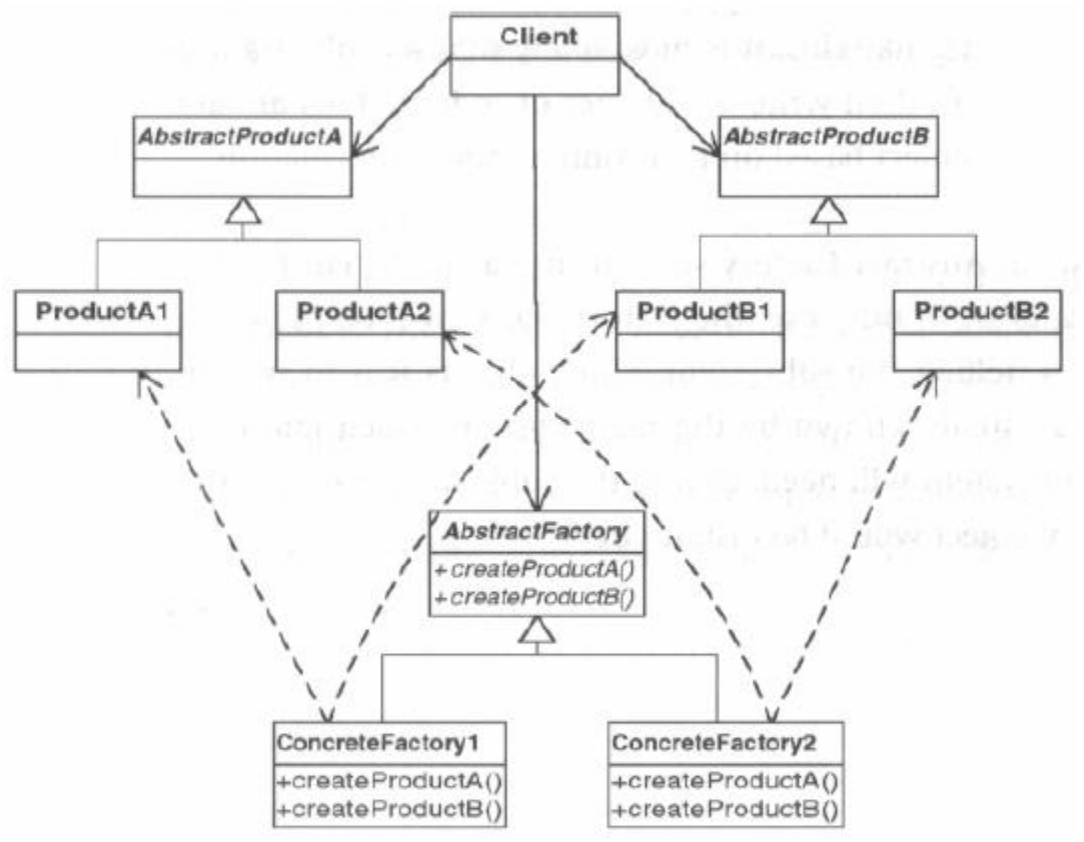
Theory:

The Abstract Factory Pattern

According to the Gang of Four, the intent of the Abstract Factory pattern is to;

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

Intent- You want to have families or sets of objects for particular clients (or cases).
Problem- Families or related objects need to be instantiated.
Solution- Coordinates the creation of families of objects. Gives a way to take the rules of how to perform the instantiation out of the client object that is using these created objects.
Participants and Collaborators- The **AbstractFactory** defines the interface for how to create each member of the family of objects required. Typically, each family is created by having its own unique **ConcreteFactory**.
Consequences- The pattern isolates the rules of which objects to use from the logic of how to use these objects.
Implementation- Define an abstract class that specifies which objects are to be made. Then implement one concrete class for each family. Tables or files can also be used to accomplish the same thing.


Client
AbstractProductA
AbstractProductB
ProductA1
ProductA2
ProductB1
ProductB2
AbstractFactory
+createProductA()
+createProductB()
ConcreteFactory1
+createProductA()
+createProductB()
ConcreteFactory2
+createProductA()
+createProductB()

Exercise:

Work in the group of three people and consider the following **factory method** example where there is class of online bookstores that can choose different book distributors to ship the books to the customers. In this example, BookStoreA and BookStoreB choose which distributor EastCoastDistributoror MidWestDistributor or WestCoastDistributor) to use based on the location of the customer. This logic is in each
bookstore's GetDistributor method. Extend this **factory method** pattern to the **abstract factory pattern** by:

1. Adding another product that the factories can produce. In the above example, we will add Advertisers that help the bookstores advertise their stores online.
2. Each bookstore can then choose their own distributors and advertisers inside their ownGetDistributor and GetAdvertiser method.

Also justify the use of abstract factory pattern by explaining its key features.