**- Creational patterns:**

- Abstract Factory

- Builder

- Factory

- Prototype

- Singleton

**- Behavioral patterns:**

- Chain of responsibility

- Observer

- Strategy

**- Structural patterns:**

- Composite

- Façade

- Proxy

- Decorator

**- Creational patterns:**

- Patterns provide a way to create objects while hiding the creation logic, Ruther than instantiating objects directly using the new operator.

**- Abstract Factory (Factory of factories):**

**- Intent:** offering an interface to create a family of related objects without explicitly specifying their classes.

**- When to use:** A system should be configured with one of multiple families of products and should be independent of how its products are created

**- Motivation:** Creating multiple factories if products have slight difference.

**- Diagram:**

****

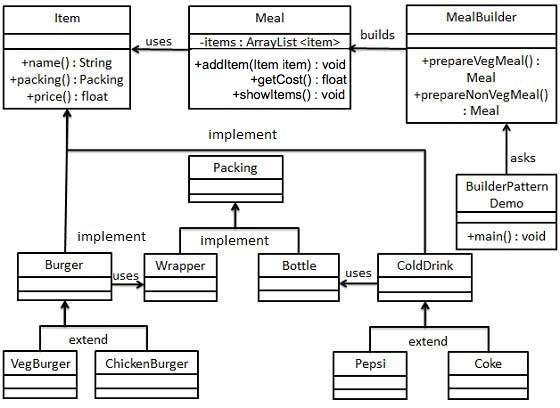
- In the diagram the factories that can be created are *ShapeFactory* and *RoundedShapeFactory* and the class responsible for creating these factories is the *FactoryProducer* basically factories in this pattern are treated as products in the *factory pattern*.

* **Builder Pattern:**

**- Intent:** Define an instance for creating an object but letting subclasses decide which class to instantiate.

**- Motivation:** creation of a complex object using simple objects and using step-by-step approach.

**- Diagram:**

****

- In this diagram the *ColdDrink* is packed in bottle and *Burger* is wrapped in a wrapper and both are packed in *Packing* then class Item is using by both items (burger and cold drink) then the *Meal* class uses the *Item* class to add items into meal then the *MealBuilder* which is the builder class uses all previous classes to build a complex meal from simple burger and cold drink.

* **Factory pattern:**

**- Intent:** Create objects without exposing the creation logic to the client.

**- Motivation:** Creation of objects derived from a common superclass to the factory.

**- Diagram:**

****

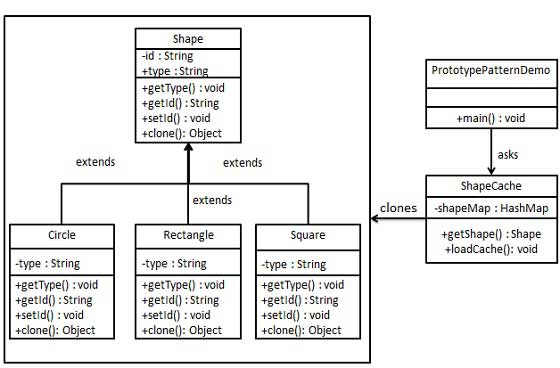
- In this diagram the shapes implement the interface and the *ShapeFactory* is responsible for creating shapes using the *Shape* interface.

* **Prototype Pattern:**

**- Intent:** Specifying the kind of objects to create using a prototypical instance. Creating new objects by copying this prototype.

**- Motivation:** This pattern is used when creation of object directly is costly**.** For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

**- Diagram:**



- In this diagram we clone the cached shape and use this clone as creating a shape directly will cost some time.

* **Singleton pattern:**

**- Intent:** Creation of a single instance of a class.

**- Motivation:** If a system has a database connection this must be only one instance of the connection so the class responsible for the connection should be applied according to the singleton pattern.

**- Diagram:**

****

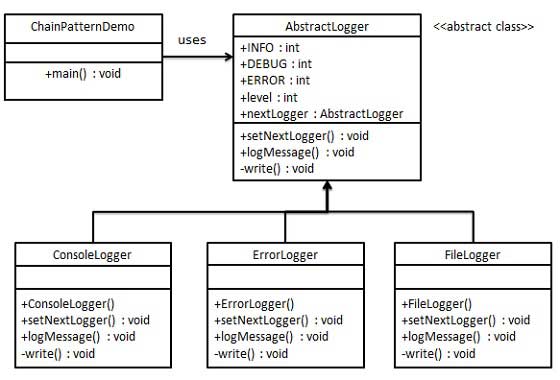
**- Behavioral Patterns:**

- These patterns are specifically concerned with the communication between objects.

**- Chain of responsibility pattern:**

**- Intent:** Pass down a responsibility among other classeswhich can share the same responsibility.

**- Diagram:**

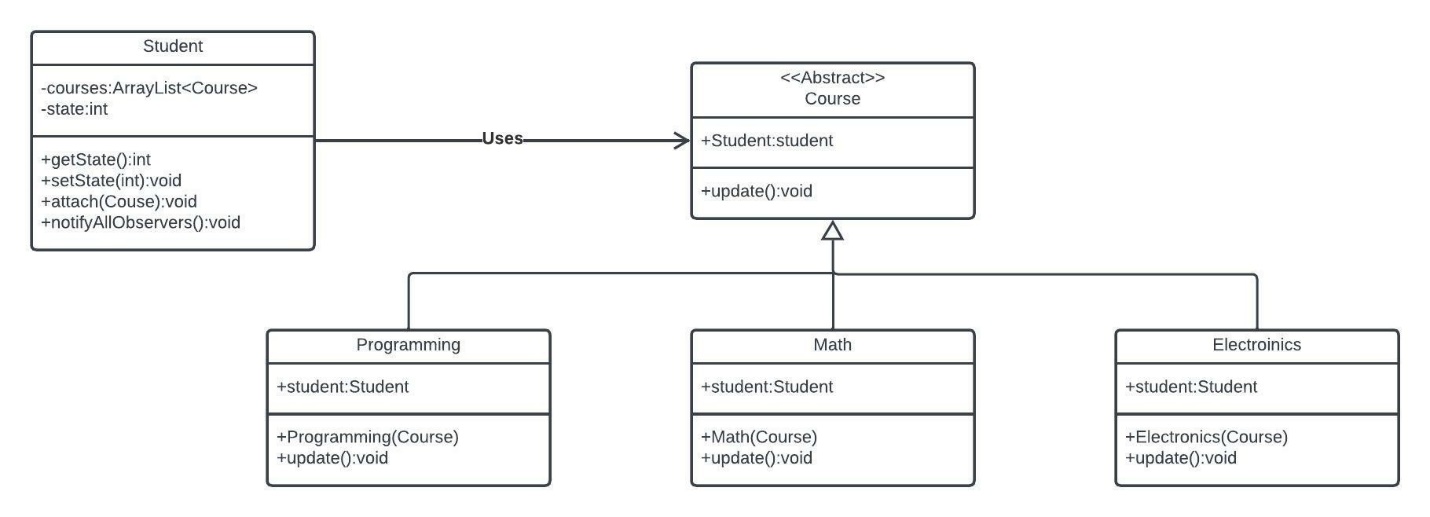
****

**-** In this diagram the *AbstractLogger* is used to hand down responsibilities among other loggers which is to write some sort of message if something happened.

* **Observer pattern:**

**- Intent:** Defines a one-to-many dependency between objects so that when an object change state all other depending objects are notified and updated.

**- Diagram:**

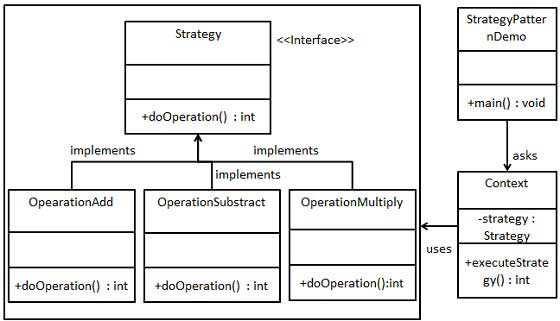
****

**-** In this diagram the observable is the abstract *Course* which is extended by other courses and the observer is the *Student* which has a list of course that he wants to be updated when any of them is changed.

* **Strategy pattern:**

**- Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable.

**- Diagram:**

****

**-** In this diagram the *Strategy* interface is implemented by multiple strategies and depending on the *Context* the strategy is chosen.

**- Structural patterns:**

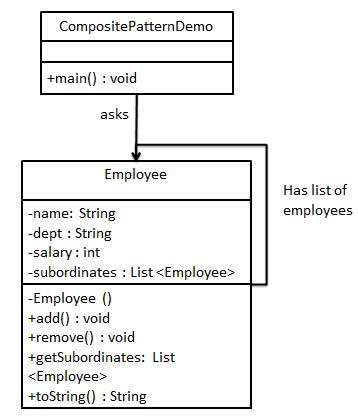
- These patterns concern classes and objects composition. Concept of inheritance is used to compose interfaces and define a way to compose objects to obtain new functionalities.

**- Composite pattern:**

**- Intent:** To compose objects into tree structure and represent part-whole hierarchies.

**- Motivation:** When a class has a group of its objects.

**- Diagram:**

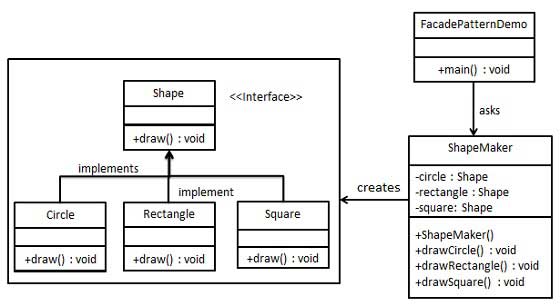


- In this diagram the *Employee* has a list of employees in itself as the employee position in an enterprise could change from one another.

**- Façade pattern:**

**- Intent:** Hide complexity of the system and provide an interface to the client.

**- Diagram:**

****

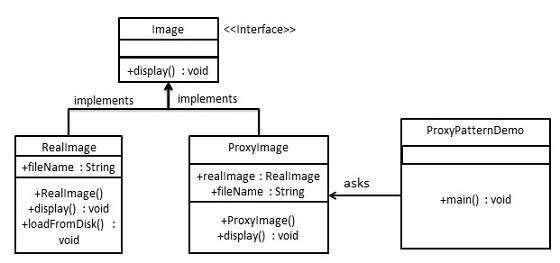
**-** In this diagram the *ShapeMaker* is the façade provided for the client to use instead of creating shapes using the *Shape* interface and new operator just make an instance of the *ShapeMaker* and use it to draw shapes.

**- Proxy pattern:**

**- Intent:** provide a placeholder for an object to control references to it.

**- Motivation:** Controlling when to load an object as the process of loading an object might cost much resources so if an object is already loaded we use its methods though an interface and if not loaded we load the original object to the interface and then use the interface.

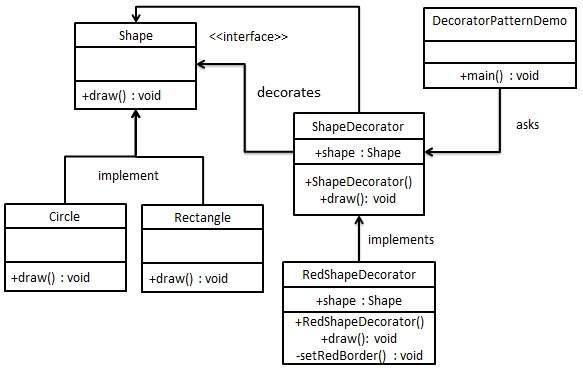
**- Diagram:**

****

**-** In this diagram the *image* interface is used to display an image the *realImage* is used by the *proxyImage* to load the real image from the disk if the image is already loaded the *proxyImage* class can use the *RealImage* methods if not the *proxyImage* load the *RealImage* and then use its methods.

**- Decorator pattern:**

* **Intent:**
  + To add additional responsibilities dynamically to an object.
* **Motivation:**
  + Creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
* **Diagram:**



- In this pattern the shape interface is used by the decorator class to add the decoration to it and any new decoration needed will extend the ShapeDecorator as shown in the diagram